Redundant Number Systems for Optimising Digital Signal Processing Performance in Field Programmable Gate Array

William Hermanus Michael Kamp B.E.(Hons.)

A thesis presented for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering
at the
University of Canterbury,
Christchurch, New Zealand.

January 2010

ABSTRACT

Speeding up addition is the key to faster digital signal processing (DSP). This can be achieved by exploiting the properties of redundant number systems. Their expanded symbol (digit) alphabet gives them multiple representations for most values. Utilising redundant representations at the output of an adder permits addition to be performed without carry-propagation, yielding fast, constant time performance irrespective of the word length. A resource efficient implementation of this fast adder structure is developed that re-purposes the fast carry logic of low-cost field programmable gate arrays (FPGAs). Experiments confirm constant time addition and show that it outperforms binary ripple carry addition at word lengths of greater than 44 bits in a Xilinx Spartan 3 FPGA and 24 bits in an Altera Cyclone III FPGA.

Redundancy also provides other properties that can be exploited for performance gain. Some redundant representations will have more zero-symbols than others. These maximise the opportunities to exploit the multiplicative absorbing and additive identity properties of zero that when exercised reduce superfluous calculations. A serial recoding algorithm is developed that generates a redundant representation for a specified value with as few nonzero symbols as possible. Unlike previously published methods, it accepts a wide specification of number systems including those with irregularly spaced symbol alphabets. A Markov analysis and analysis of the elementary cycles in the formulated state machine provides average and worst case measures for the tested number system. Typically, the average number of non-zero symbols is less than a third and the worst case is less than a half.

Further to the increase in zero-symbols, zero-dominance is proposed as a new property of redundant number representations. It promotes a set of representations that have uniquely positioned zero-symbols, in a Pareto-optimal sense. This set covers all representations of a value and is used to select representations to optimise the calculation of a dot-product.

The dot-product or vector-multiply is a fundamental operation in DSP, since it is employed in filtering, correlation and convolution. The nonzero partial products can be packed together, substantially reducing the calculation time. The application of

redundant number systems provides a two-fold benefit. Firstly, the number of nonzero partial products is reduced. Secondly, a novel opportunity is identified to use the representations in the zero-dominant set to optimise the packing further, gaining an extra 18% improvement. An implementation of the proposed dot-product with partial product packing is developed for a Cyclone II FPGA. It outperforms a quad-multiplier binary implementation in throughput by 50% .

Redundant number systems excel at increasing performance in particular DSP subsystems, those that are numerically intensive and consist of considerable accumulation. The conversion back to a binary result is the performance bottleneck in the DSP algorithm, taking a time proportional to a binary adder. Therefore, redundant number systems are best utilised when this conversion cost can be amortised over many fast redundant additions, which is typical in many DSP and communications applications.

CONTENTS

PREFACE

The work in this thesis originally began at Tait Electronics Ltd where I worked as a summer research student under the supervision of Dr Ian McLouglin. At the time the research group was developing a MIMO (multi-antenna at transmitter and receiver) radio product (branded MiMOmax) where the extensive digital signal processing is performed in a low-cost field programmable gate array (FPGA) device. Their interest in redundant number representations was in performing certain calculations faster than was currently achievable in their chosen FPGA.

When my term of employment at Tait ended at the beginning of 2006 I continued this research as a Masters research project at the University of Canterbury under the academic supervision of Dr Andrew Bainbridge-Smith. This period of 18 moths focused on the development of particular number systems and the associated arithmetic operations. After writing two thirds of a Masters thesis, we found there were still many unanswered questions and avenues to explore. So in mid 2007, encouraged by my father and supervisor, I transferred to the Ph.D. program. The scope of the project was broadened to generalise the previous work to consider a wider range of number system specifications. This resulted in improved designs and a greater understanding of the inner-workings of redundant number systems.

In writing this thesis I have assumed that the reader has an understanding of binary arithmetic and digital logic. I hope that reading this thesis will enlighten you to the alternative forms of computer arithmetic, and make you aware that two's complement binary is not the only, or always the best, solution. A range of possibilities do exist, that can lead to improved digital signal processing performance.

SUPPORTING PUBLICATIONS

- William Kamp, Ian McLouglin, and Andrew Bainbridge-Smith. An exploration of redundant number representations in FPGA. *13th Electronics NZ Conference*, pages 63–68, 13 November 2006.
- William Kamp and Andrew Bainbridge-Smith. Multiply accumulate unit optimised for fast dot-product evaluation. *Field-Programmable Technology, 2007. International Conference on*, pages 349–352, 12 December 2007.
- William Kamp, Andrew Bainbridge-Smith, and Michael Hayes. Minimum Hamming Weight Representations for Irregular Symbol Alphabets. *16th Electronics NZ Conference*, pages 87–93, 18 November 2009.
- William Kamp, Andrew Bainbridge-Smith, and Michael Hayes. Efficient implementation of fast redundant number adders for long word-lengths in FPGA. *Field Programmable Technology, 2009, International Conference on*, pages 239–246, 9 December 2009.

PLANNED PUBLICATIONS

These papers are planned for submission in 2010 and 2011. They will be derived from the associated chapters.

- Efficient implementation of fast redundant number adders for long word-lengths in FPGA. *Special issue of the Journal of Signal Processing Systems*, 2010, Material from Chapter 5.
- Zero Dominant Set. Material from Chapter 4.
- Partial Product Packing for Optimising Dot-Product Evaluation. Material from Chapter 8.

ACKNOWLEDGEMENTS

I wish to thank everyone who has assisted me thoughout this thesis research, both academically and personally. To my supervisors, Dr Andrew Bainbridge-Smith and Dr Michael Hayes, many thanks for your guidance, support, and encouragement. Your open door policies and willingness to help at all times is much appreciated. To Dr Ian McLouglin and Group Research at Tait Electronics, thank you for starting and encouraging me down this path.

While studying as both an undergraduate and postgraduate student in the Electrical and Computer Engineering department I have enjoyed great support. The knowledge and understanding given freely by the people in the department has been of extraordinary help. Of particular help has been Steve Weddell, whom i thank here for sharing his knowledge and experience of all things FPGA and DSP related.

To my parents, Peter and Betty-Ann, thank you for your advice, for instilling in me the desire for excellence, and always supporting and encouraging me in my endeavours. To my siblings, Marcel, Stephanie and Jozef, your words of encouragement, sarcastic or not, have kept me motovated; “I’ll finish it - Yeah right”. A special thanks to Kim. Your love and support means a lot to me.

To my friends, thanks for the distractions: hockey; mountain biking; snow boarding; BBQs and movie watching. You have all helped me to keep a fresh mind and to remain sane.

I wish to acknowledge and thank you the Electrical and Computer Engineering Department for funding my Doctoral Scholarship, Tait Electronics together with FRST for a TIF scholarship and the University of Canterbury for a Masters Scholarship. Thank you again to my supervisors for funding excursions to present at conferences in far-off places.

Chapter 1

INTRODUCTION

The performance and cost of digital signal processing (DSP) is constantly under pressure. For 40 years improvements in circuit technology have been relied upon to provide increases in performance and reductions in cost, while the arithmetic circuits that are implemented have not fundamentally changed. Unfortunately, the current integrated circuit technology, complementary metal-oxide semiconductor (CMOS), is reaching fundamental quantum-physical barriers [Powell, 2008]. One path to increased performance is to change the computational architectures at a fundamental level. Alternate number systems exhibit additional properties that do no exist in the binary number system. These properties may be exploited to optimise the performance of computer arithmetic.

The objective of the research presented in this thesis is to accelerate the fundamental arithmetic operations and algorithms central to DSP. This is achieved by redefining the way numbers are represented in a computer system by introducing redundancy in number representation. This redundancy gives a number of interesting properties such as an increase in the number of zero-digits. The redundancy and its properties are exploited to create fast constant time adders for integers of any magnitude, to minimise the number of partial products in a multiplication, and to reorganise calculations to maximise the productive calculating time of the logic in a multiply accumulate function.

Where re-programmability is desired and digital signal processors can no longer keep pace, equipment manufacturers have turned to field programmable gate arrays (FPGA) to exploit parallelism and pipelining in their algorithms. Even with these advantages, increasing data rates require ever higher clock rates and hence the use of the fastest and consequently the most expensive FPGA.

The target technology for implementation in this thesis is *low-cost* FPGAs such as the Xilinx Spartan and Altera Cyclone families. These devices provide a comparable amount of user-definable logic resources to a high-end FPGA for a small fraction of the cost. While the specific implementations are targeted to the low end FPGA much of the work can be applied to high end FPGAs or fully custom hardware designs in

application specific integrated circuits (ASICs). Many of the ideas presented are also technology independent, keeping them relevant for the computing circuit technologies of the future.

In particular, this thesis considers how the properties of redundant number systems can be exploited to improve the performance of DSP algorithms and how calculations using these numerical representations can be implemented efficiently in low-cost FPGAs with four-input look-up-table logic.

1.1 DIGITAL SIGNAL PROCESSING

A signal represents information about the state of an object. The signal's origins may be natural or man-made. Often the signal varies in time and/or space and the desired information is not conveyed directly by the signal. The information may be obscured by noise or interfering signals from other objects. Therefore, some processing of the signal, possibly in combination with other signals, is required to extract the information about the object's state. More commonly this processing is done using digital computer arithmetic.

A signal may be represented by an integer variable of finite range by uniformly sampling and quantising measurements of its continuous and instantaneous value [Ambardar, 1999]. This discrete, 'digital' form of the signal is suitable for mathematical manipulation using computer arithmetic. This is termed digital signal processing. The purpose is to enhance, store, analyse, modify, transmit, or extract information from the signal, possibly in combination with other related signals.

The benefits of DSP over processing signals with analogue electronics are numerous [Ifeachor and Jervis, 2002], since:

- The processing steps are exact, giving perfect reproducibility over the entire lifetime of the processor, and over wide environmental ranges of temperature and electrical interference.
- Functions can be performed that are impractical or impossible in analogue electronics since DSP is largely free from unexpected non-linear distortions.

DSP has some disadvantages that are diminishing as technology improves and subsequently performance increases. The signals that can be processed are limited by their band-width, being the difference between the lowest and highest frequency component. This is typically smaller for DSP systems than for their analogue counterparts. The

DSP bandwidth is constrained by the rate a processor can execute its algorithms. At a fundamental level this is limited by the time to execute arithmetic operations.

DSP has many fields of application including audio, communications, radio, imaging, video, control, radar, sonar, geographical positioning systems (GPS), biomedical devices, and instrumentation. The continued demand for improved fidelity and bandwidth in DSP systems requires higher performance. Higher data rates are always desired. Communications systems have been the traditional driver for increased performance, especially radio systems where the digitisation of the radio signal continues to be pushed closer to the antenna. Ultimately, this gives rise to software defined radio where the DSP completely defines the radio and it can be reprogrammed to switch between different modulation techniques and protocols [Blossom, 2004].

A DSP system can generally be separated from a general purpose computer by the type of data that it processes and the style of algorithms employed. Firstly, the data feeding a DSP system is usually a stream, or more often multiple streams of data, being continuously fed with samples at regular intervals. This requires a large amount of data movement within the processor. A general purpose computing system usually operates on blocks of data and consequently the bandwidth requirement is usually less. Secondly, DSP algorithms tend to be mathematically based and operate continuously. They are designed to modify a specific type of signal and perform the same steps repeatedly with each new sample. The algorithm's parameters may be changed to achieve a different output signal. General purpose computing algorithms tend to be more decision oriented, choosing to operate on data sets in different ways based on the data itself [Kester, 2003].

The demand for performance from digital signal processing systems continues to grow. For example, a worldwide project, the SKA (square kilometer array) radio telescope, will require an extraordinary amount of signal processing, mostly to be performed digitally and in real time [D'Addario et al., 2007]. The entire signal processing system is being re-examined including the algorithms used for correlation and the technology that they are implemented in.

1.2 ALTERNATE NUMBER SYSTEMS

Modern computer arithmetic and DSP systems almost exclusively employ the binary two's complement number system to represent integers. Its advantage is in its simplicity. Firstly, its two digits (bits), 0 and 1 map directly to the two stable electronic states of low (L) and high (H) voltage in CMOS logic circuits. A single wire transmits this information. Secondly, arithmetic operations are simple because the bits are 0, the multiplicative absorbing element and additive identity, and 1, the multiplicative identity.

Because there are only two bits, the implementing circuits do not have to accommodate many input or output cases. Thirdly, negative and positive numbers are handled uniformly. These advantages make the circuits relatively fast and cheap to produce.

A number system is said to be redundant when there are multiple representations for some or all values. Redundant number systems offer an alternate form of computer arithmetic suited to numerically intensive applications. These include FIR and IIR filtering, cryptography (such as RSA), and control applications requiring matrix multiplication. The advantages of redundant number systems come to the fore where there is significant accumulation. Performance improvements, at a basic arithmetic level and at an algorithmic level, can be made by exploiting the multiple representations and the properties that the redundancy exposes.

Redundant number systems depart from the usual binary two's complement designs for computer arithmetic. Since the result is not restricted to have a unique representation like binary, the adder can give a result representation that makes the addition faster. In the addition of two positional numbers, the digits in corresponding positions are added. If their sum cannot be represented by a digit then the sum is split into a carry digit and the remainder, where the remainder must belong to the set of accepted digits. The carry digit is added to the sum in the next position. This may cause the next position to also generate a carry. Addition with binary, and generally with non-redundant number systems, must provide a path for the carry to propagate from the least significant position to the most significant position. Propagating the carry along the addition can be a long and time consuming process. In the worst case, carry propagation occurs along the entire addition length. This is usually the critical-path, limiting the addition speed. When the result is allowed to be a redundant number, the addition can be arranged so that the carry is captured in the next position, effectively removing the propagation of a carry. Therefore, the addition can be completed much faster and the time is independent of the operand lengths.

A second advantage of redundant number systems is that some representations of an integer have a greater number of zero digits. This is an advantage in algorithms like multiplication because it reduces the number of calculations [Dempster and Macleod, 2004]. For example, in the long multiplication algorithm whenever there is a zero digit in the multiplier operand, the corresponding partial product is also zero. Since this contributes zero value to the partial product accumulation it can be skipped, saving an addition.

These advantages are at their peak in numerically intensive applications, such as DSP algorithms, and when the word-lengths of the numbers are large, like in cryptography. Applications where there is substantial accumulation benefit hugely from

the constant time addition properties of redundant number systems. An example is the dot-product/vector-multiply operation used in many DSP algorithms for filtering, convolution, correlation and beam-forming. The dot-product operation is the sum of many pair-wise multiplications, where each multiplication is a sum of partial products. The dot-product can also benefit from increasing the number of zero digits to reduce the number of partial product additions.

Despite these advantages, redundant number systems are not in common use. The added complexity, area usage and design time required to use an alternate number system has not been worth the performance increase. Simply waiting for integrated circuit (IC) technology to improve has been sufficient, with the clock rate doubling with the doubling of transistor density approximately every 24 months as predicted by Moore [1965]. However, clock rates in processors have hit a ceiling. To maintain Moore's law now requires too much power in too small an area [Balandin, 2009]. The subsequently generated heat cannot be removed fast enough and ICs simply become too hot. To continue Moore's law in terms of increasing computational performance, microprocessor designers are putting multiple cores on a single IC [Hill and Marty, 2008].

Redundant number systems are not particularly suited to the implementation of a generic arithmetic logic unit (ALU), a main component of a CPU. It is difficult to define and implement the Boolean operations of NOT, AND, OR, and XOR such that they give an equivalent result to binary. This is simplified for a binary ALU because the two's complement binary number system coexists with logical bit vectors. Redundant number systems are better suited to hardware designs of specific co-processing blocks such as the dot-product.

1.3 EXISTING AREAS OF RESEARCH

With unlimited combinations of radicies and symbol alphabets the space of possible number systems is infinite, as lamented in a number of papers on redundant number systems [Odlyzko, 1978, Parhami, 1990, Arno and Wheeler, 1993]. Thus many researches have chosen to focus on very restricted symbol alphabets and small radicies, usually the binary radix. Their symbol alphabets are designed to facilitate some property they wish to exploit, commonly carry-propagation free addition [Avizienis, 1961, Jaberipur and Parhami, 2008, Kawahito et al., 1990, Parhami, 1988, Harata et al., 1987, Jaberipur et al., 2001] or the minimum Hamming weight property [Xu et al., 2007, Koc and Hung, 1992, Arno and Wheeler, 1993, Jedwab and Mitchell, 1989] as discussed in Section 2.4 and further developed in Chapter 3.

In the construction of fast multipliers it is beneficial to have as many zeros as

possible in the multiplier representation. There have been many suggestions for recoding algorithms to build redundant representations of integers for this purpose. A first attempt was suggested by Booth [1951]. This was further expanded to exponentiation [Jedwab and Mitchell, 1989, Egecioglu and Koc, 1994, Lou and Wu, 2004], particularly in the application of public key cryptography [Gordon, 1998]. One of the most established classes of redundant representations for this purpose is NAF (non-adjacent form) introduced by Reitwiesner [1960]. NAF is so named because at most one out of every two consecutive symbols is nonzero. Reitwiesner also showed that each integer had a unique NAF representation, thus it is also often called the canonical form [Coleman and Yurdakul, 2001, Koc and Hung, 1992, Egecioglu and Koc, 1994]. It has been applied to Montgomery multiplication methods [Kaya Koc et al., 1996] used in, for instance, the RSA encryption algorithm, and to elliptic curve scalar multiplication [Morain and Olivos, 1990] used in, for example, the Diffie-Hellman key exchange.

The number of guaranteed consecutive zero-symbols can be increased by extending the symbol alphabet to include values in the range $(-2^{w-1}, 2^{w-1})$. This gives the width-w non-adjacent form (w-NAF) extending the property of minimum Hamming weight as described in [Muir and Stinson, 2004]. Implementing cryptographic systems in smart-card devices with minimal memory has prompted the development of left-to-right recoding of binary representations into w-NAF representations [Muir and Stinson, 2005], or other similar representations such as the mutually-opposite form (MOF) [Khabbazian et al., 2005], eliminating the need to store the long multiplier operand.

Most reported practical designs using redundant number systems are implemented in VLSI silicon using a CMOS process. Many of these designs are for multipliers [Takagi et al., 1985, Ercegovac and Lang, 1990, Chang and Parhi, 1998], dividers [Kuninobu et al., 1987b], exponentiation and square-root [Oklobdzija and Ercegovac, 1982].

There is an interesting body of work in multiple constant multiplication (MCM) and sub-expression sharing [Boullis and Tisserand, 2005, Kharrat et al., 2002, Voronenko and Püschel, 2007]. The primary focus of applying redundant number systems, primarily binary signed digit, is the reduction in Hamming weight for the coefficients. This means fewer additions or subtractions are required. A related method presented by Samueli [1989] for the design of finite impulse response (FIR) filters is called 2PFIR. Here the FIR coefficients are chosen from a set of legal values that are those which can be encoded with a Hamming weight of two or less.

Beyond the fixed integer radix are some more exotic number system suggestions. These include a complex-base number system using the radix $2i$ [Knuth, 1960, Holmes, 1978], Fibonacci coding, and the golden-ratio base with the binary symbol alphabet [Bergman, 1957].

1.4 COST, ENERGY, AND DELAY

Three interrelated factors are used to evaluate and compare computer systems. These are cost, energy, and delay or alternatively the three Ps: price, power, and performance. They are not independent, improving one usually detrimentally affects the other two [Nagendra et al., 1996]. In the design of a system, these three measures are traded off against each other in an attempt to reach the perfect system that costs nothing, uses no energy, and performs its function in zero time.

Cost indicates the difficulty in manufacturing an integrated circuit. The manufacturing process is not perfect, process variation and random defects in the silicon wafers results in below specification or defective circuits [Mangir, 1984]. The manufacturing yield is the proportion of functional circuits. A circuit that occupies less silicon area is less likely to contain a defect; giving a higher yield. As the complexity of a circuit increases it occupies greater silicon area decreasing the number of devices per wafer and increasing the probability of a faulty device. A larger silicon area translates to a lower yield of fully functional devices, hence a higher cost per chip. Therefore, cost is strongly correlated with the resource usage of a design i.e., its area. The reconfigurable nature of FPGAs can allow minor manufacturing variation to be mitigated to some extent by testing each device and synthesising the design around minor defects [Sedcole and Cheung, 2007]

Performance measures the throughput of a system. In the case of computer arithmetic, this is the rate of execution and can be expressed as a time elapsed (delay) or inversely as the rate of repetitive execution (frequency). The execution rate is dependent on many factors including the algorithms used for calculation, their implementation, and the rate of internal arithmetic. Many metrics try to quantify the performance of systems and their elements. For algorithms this may be the number of calculations per data input expressed as a polynomial or logarithmic expression in the quantity of data, or generally in big-O notation [Graham et al., 1994]. For arithmetic circuits the number of calculations per unit time are commonly measured in MOPS (millions of operations per second)[1]. DSP systems as a whole are often measured in terms of SPS (samples per second), being the maximum rate that the input stream of samples may be processed. Comparisons of overall system performance is best measured using benchmark tests.

Energy is the electrical energy consumed by the system to perform its function. Minimising the energy provides many benefits including maximising battery life and reducing the heat dissipation requirements. The term *power* is used to describe the energy consumed per unit time. In digital circuits, power can be split into two components;

[1] MOPS is a loose term as it avoids the definition of an operation, it is better suited to sequential software.

static power consumed by the system independent of computations taking place and the *dynamic power* used for computing. Static power is technology dependent, depending on the size and type of transistors used [Chandrakasan and Brodersen, 1995]. Dynamic power is dominated by the number of transistors and how often they switch. Switching a MOSFET[2] transistor requires the movement of charge to or from its gate capacitance, this current ultimately causes ohmic power losses [Veendrick, 1984]. To minimise dynamic power, the circuit designer needs to minimise how often the transistors switch, including glitching; the switching that occurs as the circuit logic settles to its final state. The designer should also try to minimise the number of transistors, also reducing the static power and cost [Chandrakasan and Brodersen, 1995].

1.5 HARDWARE PLATFORMS

There are many applications that are heavily dependent on DSP. These include:

1. Data and voice communications systems requiring signal processing at various stages thoughout the system, including voice processing, filtering, encryption, modulation, demodulation, and error correction.
2. Photo, video, and audio recording, storage and reproduction for media applications.
3. Sensor measurement for applications from weather stations to aircraft flight control, to scientific experiments.

The applications have widely varying demands for price, performance and power. Mobile communications systems are some of the most demanding applications with requirements for low latency, real time processing at high data rates, while consuming little power.

To meet the demands of DSP applications various arithmetic hardware platforms have been developed, including:

1. General purpose microprocessors or central processor units (CPU). CPU are not optimised for performing signal processing and are often used at the user level of systems, basic signal processing or for processing large data sets not in real time (offline).
2. Digital signal processors. Embedded processors that are optimised for DSP with special instructions, fast and parallel memory access to quickly move data and supporting peripherals that make data handling fast and efficient. These execute

[2]MOSFET - metal-oxide semiconductor field effect transistor.

sequential programs on a few signal samples at a time. This makes them extremely versatile, although relatively slow. They are most effective in low power systems and at audio data rates.

3. Graphics processing units. These are processors that specialise in image processing and are capable of quickly operating on huge data sets of similar data. These are high power devices with many parallel data paths and high memory bandwidth giving them high performance. The many parallel data paths make the silicon area large and hence the device expensive.

4. Application specific integrated circuit (ASIC). These are a fully customised design for an integrated circuit that will perform one function, specific to an application. Their performance is extremely high and energy consumption minimal, however, they have very high development costs and high fixed production costs but comparatively low per unit cost. This makes them most suitable for high quantity production or where power and performance are absolutely critical.

5. Programmable logic devices (PLD) including field programmable gate array (FPGA). FPGA are similar in versatility to ASIC, though they have higher energy usage and lower performance. A key advantage of FPGAs their reprogrammability. This makes prototyping much simpler; reducing the system development cost. The per unit cost is moderate, making them most suited to low to medium production quantities of high performance systems, or where reconfigurability is desireable. Their use in signal processing is growing where the quantities do not warrant the use of ASICs, often in niche communications and signal processing applications.

FPGAs are chosen as the development platform in this thesis. Although the designs to be presented in this thesis are targeted at the low-cost Altera Cyclone III and Xilinx Spartan 3 FPGAs, they should translate easily to ASIC designs and other FPGA families.

1.6 FIELD PROGRAMMABLE GATE ARRAY

A field programmable gate array (FPGA) is an integrated circuit that can be configured *after* manufacture i.e., programmed in the field. The fundamental architecture has a regular structure of small circuits referred to as logic blocks, with wires that interconnect these circuits; this is the array of gates. Each logic block can be programmed to provide a combinational logic function and one bit of memory. Larger logic functions can be created by connecting the logic blocks' outputs and inputs together using the programmable interconnect wires. Thousands of logic elements exist in a single FPGA

allowing it to implement many different and complex logic functions. Any digital design can be implemented in a suitably sized FPGA including entire microcontrollers [Altera Nios II, 2009, Xilinx Microblaze, 2008].

FPGA devices are implemented using static random access memory (SRAM) technology, providing the field programmability. The SRAM is volatile so after a device is powered off, the stored logic configuration is lost. This is in contrast to other programmable logic devices such as PAL[3] and GAL[4] that use fuses or EEPROM[5] to set the configuration. To restore a logic configuration to an FPGA, a configuration file is loaded from a (usually off chip) non-volatile memory or host processor connection. This provides an additional dimension of flexibility not available in ASICs. The logic configuration can be changed whenever necessary by loading a different configuration file. This reconfiguration allows for fast prototyping and makes FPGAs a powerful and complementary component to a microprocessor that can provide hardware accelerated functions.

The two main manufacturers of FPGAs are Xilinx and Altera who in 2008 shared approximately 86% of the reconfigurable logic market [Altera Corporation, 2008]. Other competitors include Lattice Semiconductor, Actel, SiliconBlue Technologies, and Achronix; each of which focus on a niche market such as low power, mixed signal, or NAND flash memory based FPGA. The work presented in this thesis focuses on the low-cost spectrum of FPGA. In particular, the low level architectures of the Xilinx Spartan 3 and the Altera Cyclone III will be exploited.

The low-cost FPGAs have a four input look-up table (4-LUT) as the basic combinational logic generating function, capable of implementing any four to one logic function. Traditionally, a 4-LUT is employed because this configuration has the lowest area-delay product based on benchmark designs [Kuon and Rose, 2008]. The more expensive families of FPGA, such as the Altera Stratix 3 [Altera Stratix III, 2009] and Xilinx Virtex 5 [Xilinx Virtex-5, 2009], provide six input look-up tables (6-LUT). The larger LUTs improve performance because larger than 4-input logic functions do not have to enter the interconnect network at intermediate stages where the bulk of the path delay is accumulated. This is at the expense of increased silicon area, since a 6-LUT is four times larger than a 4-LUT. To regain some area efficiency, these 6-LUTs can typically be configured as two smaller LUTs [Altera Stratix III, 2009, Xilinx Virtex-5, 2009]. The logic blocks and interconnecting wires are commonly termed the resources of a FPGA.

[3]PAL - programed array logic.
[4]GAL - gate array logic.
[5]EEPROM - electrically erasable programmable read only memory.

1.7 THESIS OVERVIEW

The work in this thesis covers a diverse range of topics. The thesis has been structured so that the ideas in one chapter build on those of previous chapters. Within each chapter the background material specific to the chapter's contents is presented first, followed by the original work.

Chapter 2 looks in more detail at number systems, in particular positional number systems and how numbers can be represented. Redundant number systems are fully introduced along with their property of Hamming weight and the ranges of values that can be represented while satisfying different conditions. An algorithm is given that generates all possible redundant number representations for a given value. Mapping the redundant number system to digital logic is also considered.

Chapter 3 examines the importance of the number zero and the minimum Hamming weight property of redundant number systems. A method of constructing representations with the minimum Hamming weight property is studied, although it only works for a limited variety of number systems. To investigate the minimum Hamming weight property further, a new algorithm was developed to generate minimum Hamming weight representations for an arbitrary redundant number system specification.

Chapter 4 extends the minimum Hamming weight property in a novel way by acknowledging that the position of a zero digit in a representation is also significant. It defines zero-dominance; a new property of redundant representations. An algorithm is developed to generate all zero-dominant representation of a value, the set of which is inclusive of the set of minimum Hamming weight representations.

Chapter 5 studies addition, the fundamental operation in computer arithmetic, and how it is implemented in FPGAs. The "carry-free" nature of redundant representation addition is presented, along with methods by which it may be achieved. A resource-efficient adder implementation for redundant representations is developed that closely matches the architectures of the low-cost FPGAs. It utilises the FPGAs' additional logic paths intended for ripple-carry addition. The performance of these adders is tested and compared to common binary adder implementations.

Chapter 6 concerns the conversion between the commonly used binary system and redundant representations. A parallel encoder is developed to convert binary to a minimum Hamming weight redundant representation in constant time irrespective of the word length, making it suitable for augmenting parallel multiplication. Methods for conversion from redundant representations to a unique non-redundant representation is described. The detection of zero value or the sign of a redundant representation is also

studied. Combining a subtractor with sign detection leads to a comparison operation.

Chapter 7 investigates algorithms that incorporate shifting and how this relates to redundant representations. To take advantage of the minimum Hamming weight property, selectable shifters are used within serial and parallel multiplier architectures. Division by shifting is considered, particularly focusing on the problems associated with truncating redundant symbols from the least significant positions.

Chapter 8 develops a novel method called partial product packing to compress the calculations in a dot-product. Combinatorial optimisation methods are used to choose zero-dominant representations for the coefficients. This effectively reduces the number of traditional multiplications to the average minimum Hamming weight of the redundant number system used.

Chapter 9 develops a dot-product unit that facilitates the partial product packing and is suitable for calculating FIR filters. A realisation of a hardware multiply accumulate unit using redundant representations is described. Its performance shows an improvement over a binary implementation with four parallel multipliers.

Chapter 10 looks at some practical constraints on the symbol alphabets and radix. Using the results from previous chapters, a selection of redundant number systems are analysed to determine the best redundant number representations to use in various situations.

Chapter 11 draws some conclusions and suggests some further avenues to explore to extend this work both in applications and number system theory.

Chapter 2

NUMBER SYSTEMS

> "It is India that gave us the ingenious method of expressing all numbers by means of ten symbols, each symbol receiving a *value of position* as well as an absolute value; a profound and important idea which appears so simple to us now that we ignore its true merit. But its very simplicity and the great ease which it has lent to computations put our arithmetic in the first rank of useful inventions; and we shall appreciate the grandeur of the achievement the more when we remember that it escaped the genius of Archimedes and Apollonius, two of the greatest men produced by antiquity."
>
> — Pierre-Simon Laplace

A number system defines a way to express values and to perform operations with them. A number is represented as a string of symbols that is interpreted in a consistent way to give its value. Number systems generally fall into two categories, additive or positional.

Additive number systems use a collection of distinct symbols, each with a discrete value. A string of symbols is interpreted by summing the values of the symbols. An example is tally marks, which has a single symbol (or mark, 𝍠) with a value of one. Larger values are recorded by appending more 𝍠 symbols, for example, 𝍠𝍠𝍠 represents three. Commonly every fifth mark is distinguished to make counting easier by striking though the previous four marks to group them, e.g. 𝍪, making a second symbol meaning five. Another common example is roman numerals with symbols I=1, V=5, X=10, L=50, C=100, D=500, and M=1000. The interpretation of values is complicated by also using subtraction based on the relative location of symbols. For example, a small symbol following a large symbol means add, so XI = 10 + 1 = 11, and a small symbol before a larger one means subtract, so IX = 10 - 1 = 9. Problems with additive system include:

1. The value of zero cannot be explicitly written down.

2. The magnitude of a value is not necessarily related to its length.

3. Very large values are difficult to express.

4. Arithmetic is difficult and impractical to perform.

Despite these disadvantages, additive systems still have usage in modern society. Tally-marks excel at being easily increased making them useful for counting. Roman numerals are difficult to drastically modify unnoticed making them useful in legal documents.

Positional number systems address the additive deficiencies. Like additive systems, positional number systems have a collection of discrete valued symbols and larger values are represented by a string of symbols. However, both the symbol's value and its absolute position in the string are significant in determining the value. Central to the positional system is a symbol with zero value. It is required as a place holder in the number string to indicate that this position contributes zero value. Positional systems are the most commonly employed in modern society for computational work and in language. The most commonly used is the decimal (base 10) system derived from the number of fingers on both hands. Positional number systems and their nomenclature are explained in Sections 2.1 and 2.2.

Redundant number systems, introduced in Section 2.3, allow a value to be expressed in many different ways. This modifies the number of zero symbols and range of a number representation as explained in Sections 2.4 and 2.5 respectively. Visualising a redundant number system and in particular all its possible representations for a value is difficult. Section 2.6 describes a way to view these as graphs of connected nodes. An algorithm for building a tree and graph of the redundant representations of a value is given in Section 2.7. Section 2.8 deals with mapping a number system to the constraints of digital logic. This exposes the practical limitations on the number system parameters. These are used to describe the number systems of interest for computing in Section 2.9.

2.1 ANATOMY OF A POSITIONAL NUMBER SYSTEM

This section describes the foundations of a positional number system and lays out the notation used throughout this thesis. The minimum required elements to define a positional number system are a radix and an alphabet of symbols. These elements are defined as:

- **Radix/Base** is the geometric factor between the place values in a positional number system. The algebraic symbol used is β.

Table 2.1: Examples of number systems with their symbol alphabet and radix.

Common Name	Symbol Alphabet ($\mathbb{S}$)	Radix (β)
Binary	$\{0, 1\}$	2
Ternary	$\{0, 1, 2\}$	3
Balanced Ternary	$\{\overline{1}, 0, 1\}$	3
Octal	$\{0, 1, 2, 3, 4, 5, 6, 7\}$	8
Decimal	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$	10
Hexadecimal	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$	16
Carry-Save	$\{0, 1, 2\}$	2
Signed Digit	$\{\overline{1}, 0, 1\}$	2

- **Symbol** is a place holder in a positional number system that has an associated value. The symbols used in the thesis are usually written using the decimal digits and associated with their usual value. If a symbol has negative value it is written with an overbar, for example, the symbol associated with the value negative one is written $\overline{1}$. The variable s is used to represent an arbitrarily valued symbol algebraically. The noun *symbol* is used, as in automata theory Hopcroft [2001], instead of digit or bit to avoid confusion with their commonly used contexts of decimal or binary respectively. Symbol is also used instead of numeral or number which have other common meanings.

- **Alphabet** is the set of legal symbols, as used in automata theory Hopcroft [2001]. An alphabet is represented in this thesis as $\mathbb{S}$ and its elements are defined within braces, for example, $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$. Its cardinality is written $|\mathbb{S}|$. In a positional number system, the alphabet will always contain the symbol 0 with zero value [1].

Some common examples of number systems are given in Table 2.1. The last two are redundant number systems.

2.2 NUMBER REPRESENTATIONS

Given a symbol alphabet $\mathbb{S}$ and radix β, a number representation can be constructed as an ordered list of N symbols, $s_{\langle N-1\rangle} s_{\langle N-2\rangle} \cdots s_{\langle 1\rangle} s_{\langle 0\rangle\ \beta}$, where $s_{\langle i\rangle} \in \mathbb{S}$. The angle brackets $\langle\ \rangle$ are used to denote the symbol's ordinal position, or rank, within the representation. N is the word-length or width of the representation. Each position

[1] This is not strictly required, an alternative is to have two symbols s_1 and s_2 such that $s_2 = -\beta s_1$. Then zero can be represented as $N/2$ continuously repeating symbol pairs, $(s_2 s_1) \ldots (s_2 s_1)_\beta = 0$ which cancel each others value.

Table 2.2: Example values for the representation 1101_β using different radicies.

Common Name	Repr	β	$s_{\langle 3\rangle}\beta^3 + s_{\langle 2\rangle}\beta^2 + s_{\langle 1\rangle}\beta^1 + s_{\langle 0\rangle}\beta^0$	Value
Binary	1101_2	2	$1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$	$= 11_{10}$
Ternary	1101_3	3	$1\times 3^3 + 1\times 3^2 + 0\times 3^1 + 1\times 3^0$	$= 37_{10}$
Octal	1101_8	8	$1\times 8^3 + 1\times 8^2 + 0\times 8^1 + 1\times 8^0$	$= 577_{10}$
Decimal	1101_{10}	10	$1\times 10^3 + 1\times 10^2 + 0\times 10^1 + 1\times 10^0$	$= 1101_{10}$
Hexadecimal	1101_{16}	16	$1\times 16^3 + 1\times 16^2 + 0\times 16^1 + 1\times 16^0$	$= 4353_{10}$
Carry-Save	1101_2	2	$1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$	$= 11_{10}$
Binary Signed Digit	1101_2	2	$1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0$	$= 11_{10}$

has associated with it a geometric weighting factor. The weighting factor is usually an integer power of the radix, β^i, where i corresponds to the rank, written in decimal. For integers, combining the symbol values and the geometric positional weights gives the representation a value v of,

$$v = \sum_{i=0}^{N-1} s_{\langle i\rangle}\beta^i. \tag{2.1}$$

The higher ordinal positions have a higher geometric weighting, hence contribute more magnitude to the value. The nonzero symbol occupying the highest position is the most significant symbol (MSS). The symbol occupying the lowest position is the least significant symbol (LSS). A representation of v is written as a string of ordered symbols from the MSS, $s_{\langle N-1\rangle}$, on the left to the LSS, $s_{\langle 0\rangle}$, on the right followed by the radix in subscript, such as $s_{\langle N-1\rangle}s_{\langle N-2\rangle}\cdots s_{\langle 1\rangle}s_{\langle 0\rangle\beta}$. If the radix is omitted then it is assumed to be ten. Examples of a representation string using different radicies is shown in Table 2.2.

A fractional number can be represented by allowing the positions to have fractional weights, that is the radix raised to a negative power. This is achieved by modifying (2.1) to start m positions below position 0, so the value v of a fractional representation is,

$$v = \sum_{i=-m}^{N-m-1} s_{\langle i\rangle}\beta^i = \beta^{-m}\sum_{i=0}^{N-1} s_{\langle i-m\rangle}\beta^i. \tag{2.2}$$

Since m is fixed, this class of representations are known as fixed point and can represent rational real numbers with a resolution of β^{-m}. The "ones" position (β^0) is usually marked with a point '.'[2] to the right of the symbol. This separates the integer and fractional parts of the representation.

[2]the decimal point is a comma ',' in Europe.

As shown in (2.2), the fractional number is simply a scaling by β^{-m} of the integer given by (2.1). Therefore, without loss of generality, only integer representations need to be considered when manipulating representations. This is the case throughout the thesis.

A negative number can be represented in several different ways:

- Implicitly through the use of an alphabet containing negative symbols or a negative radix. The negative numbers are not treated specially and arise when the negative symbols contribute greater magnitude to (2.1) than positive symbols;
- Explicitly by prepending a special sign symbol to the representation. The sign symbol modifies the interpretation of the representation's value. This can be done in a sign-magnitude way where the magnitude is evaluated as usual with (2.1), then multiplied by negative 1 if the sign is negative. Equivalently, the sign of each symbol could be reversed by a negative sign;
- Radix's-complement method using wrap-around where exceeding the most positive value in a number systems range wraps to the most negative value as shown in Figure 2.1. The MSS $s_{\langle N-1\rangle}$ carries the sign information, if $s_{\langle N-1\rangle} \geq \lfloor \beta/2 \rfloor$ then the value is negative. To interpret a negative value using (2.1), reverse the sign of all symbols except the MSS. This is the method used for the ubiquitous two's complement binary number system. For example, the 8 bit binary two's complement representation of -43 is written 11010101_2, since the MSS is 1 the number is negative and interpreted as $1\overline{1}0\overline{1}0\overline{1}0\overline{1}_2$ or $128 - 64 - 16 - 4 - 1 = -43$.

Within this thesis the implicit method will be used most frequently for representing negative numbers. The exception is that binary will usually use the two's complement method.

For a detailed review of these established topics see one of [Koren, 2002, Parhami, 2000, Knuth, 1998, Hwang, 1979].

2.3 REDUNDANT NUMBER SYSTEMS

Redundant number systems are characterised by having multiple representations for some or all representable values. There is usually a one-to-one correspondence between an integer value and a representation in a well-defined, minimal, and complete number system. That is, the number system that can represent the complete but finite integer number line, using a symbol alphabet with a cardinality of β. Therefore, the symbols

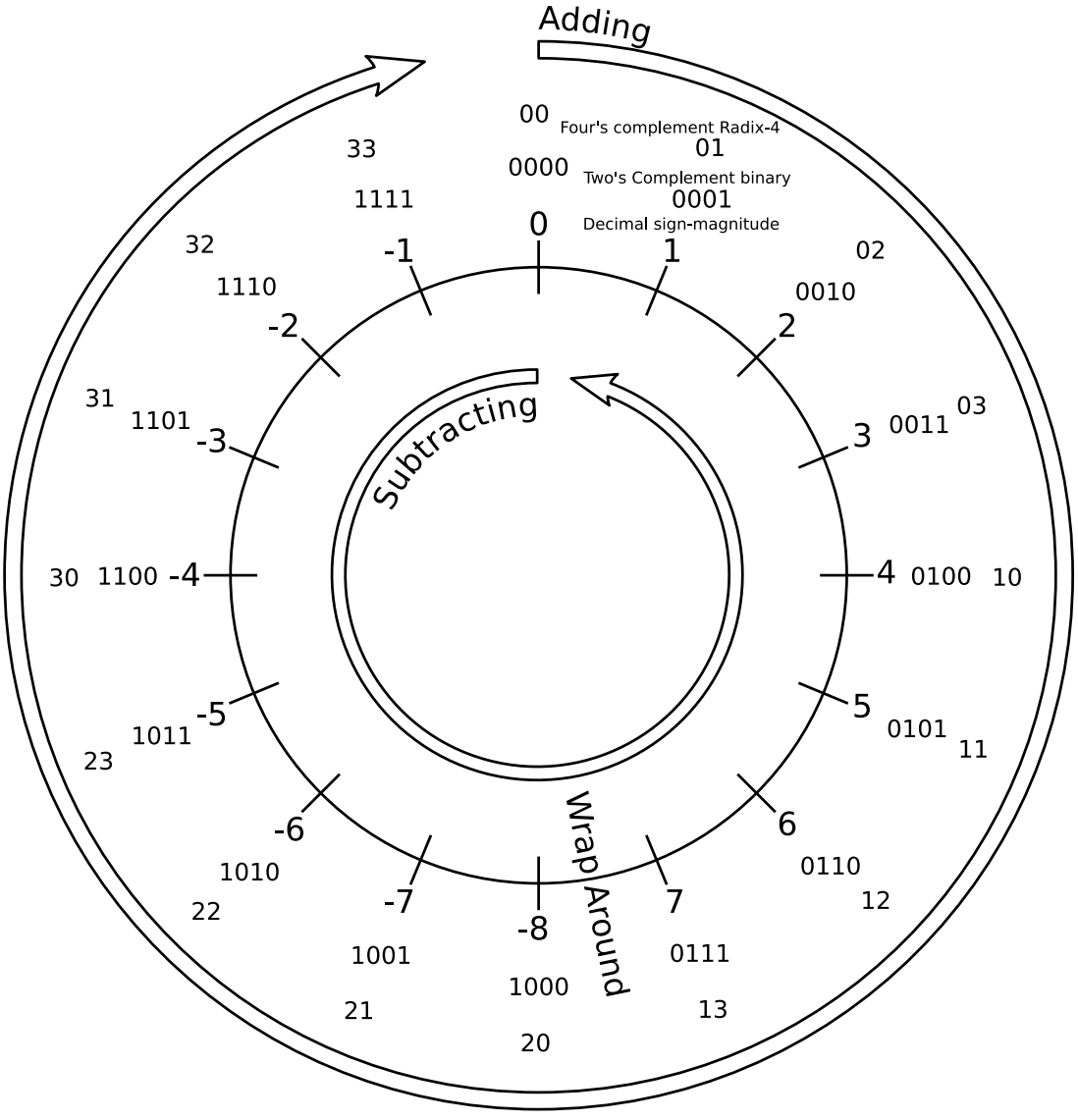


Figure 2.1: Wrapping number line with positive and negative two's complement binary and four's complement radix-4 representations.

must have the set morphism[3],

$$f : \mathbb{S} \to \{0, \ldots, |\beta| - 1\}, \tag{2.3}$$

where,

$$f(s) := s \bmod \beta. \tag{2.4}$$

The mod operator[4] returns the remainder after integer division s/β, with the result having same sign as the divisor, β. If the symbol alphabet cardinality is greater than the radix ($|\mathbb{S}| > \beta$) then the morphism of (2.3) will be overspecified with multiple symbols mapping to the same f value. This gives rise to redundant representations.

Redundant number representations for computing were first discussed by Avizienis [1961]. He introduced the signed-digit (SD) number system and showed that using it can allow addition to be performed in constant time irrespective of the word length. Usually, to perform an addition a carry signal must be propagated along the full length of the operands, taking a time proportional to the word-length. Avizienis stated that the time for addition is constant, irrespective of the operand width, if the number system used has a radix $\beta \geq 3$ and symbol alphabet $\mathbb{S} = \{\overline{\alpha}, \ldots, 0, \ldots \alpha\}$, where α is an integer such that $\beta/2 < \alpha < \beta$. This condition on the number system has gone through many revisions and is discussed in Section 5.4 on the development of carry-propagation-free addition. Section 5.5 develops a method for designing carry-propagation-free addition that places no restrictions on the operand symbol alphabet, only that the output number system must be redundant.

The number of representations a value v can have is large and can be estimated as,

$$\widehat{\#_{\text{reps}}(v)} = \frac{|\mathbb{S}|^N}{\beta^N}, \tag{2.5}$$

where $|\mathbb{S}|^N$ is the number of permutations for N symbols chosen from the symbol alphabet $\mathbb{S}$, and β^N is the approximate size of the representable integer range with N symbols. For example, a redundant number system with $|\mathbb{S}| = 3, \beta = 2$ will have of the order $(3/2)^N$ representations per value on average. Therefore, a $N = 16$ position word will have approximately $(3/2)^{16} = 656.8$ representations on average for each value. Some of these alternate representations have interesting properties. These will be identified and methods of exploiting these properties and the redundancy will be investigated throughout this thesis.

[3] meaning a function f that describes a structured relationship between two sets.

[4] This definition of mod is compatible with the python programming language, but not with C(1999+), previously undefined in C(1990). Of those languages that define both the rem and mod operators, mod is defined to have the same sign as the divisor and rem defined to have the same sign as the dividend. A collation of many programming languages' definitions of mod is available at `http://en.wikipedia.org/wiki/Modulo_operation`, *retrieved 21 January 2010.*

Using these number representations gives performance gains in some operations such as addition. However, other operations, such as sign detection, suffer a performance degradation. This is generally a consequence of the many-to-one mapping between representations and their value. Interfacing with conventional systems that use a one-to-one mapping, typically binary, requires conversion between number systems. This raises the question of which of the many redundant representations should be chosen when converting from binary. This is discussed in chapters 3 and 4, along with methods to do the conversion to a redundant number system. Methods of converting back to non-redundant representations are presented in Section 6.2.

2.3.1 Binary Signed Digit

One of the simplest redundant number systems is binary signed digit (BSD) having the symbol alphabet $\mathbb{S} = \{\overline{1}, 0, 1\}$ with corresponding values $\{-1, 0, 1\}$ and a radix of $\beta = 2$ [Parhami, 1990]. It is a redundant number system because the cardinality of the symbol alphabet $|\mathbb{S}| = 3$, which is greater than the radix $\beta = 2$. This gives most values multiple BSD representations. For example, the value 15 can be written in BSD with $N = 5$ symbol positions as 01111_2, evaluated using (2.1) as $8 + 4 + 2 + 1$, similar to binary. Other representations of fifteen in BSD include $1000\overline{1}_2$, evaluated as $16 - 1$, as well as $100\overline{1}1_2$, $10\overline{1}11_2$ and $1\overline{1}111_2$.

2.3.2 Hybrid number systems

In a hybrid number system the symbol alphabet used at each position is varied. Usually a non-redundant alphabet is used and occasionally a position is promoted to a redundant alphabet. An example of this is where most symbol positions use the binary alphabet $\{0, 1\}$ and at regular intervals the carry-save symbol alphabet $\{0, 1, 2\}$ is used [Phatak and Koren, 1994]. A hybrid number system is used to trade off the performance benefits of redundant representations with the reduced logic usage of non-redundant representations [Jaberipur and Parhami, 2008].

Mixed-radix systems vary the radix at each position in the representation. The radicies usually correspond to natural breaks in the representation of a number. Some examples are:

- Date and Time; the time Thursday 1:11:27 PM is 393087 seconds from the start of the week.

Radix	7	2	12	60	60
Denomination	Day	Half-day (AM,PM)	Hour	Minute	Second
Position (seconds)	86400	43200	3600	60	1

- Latitude and Longitude; the latitude S 43° 31' 18.65" is 804678.65 seconds from the north pole.

Radix	2	180	60	60
Denomination	Northing/Easting	Degrees	Minutes	Seconds
Position (Minutes)	648000	3600	60	1

- Currency denominations; the change received of 790 cents is made up of a \$5 note, a \$2 coin, a 50c coin and two 20c coins. Also written as $1_{2.5}1_20_21_{2.5}2_20_{10}$ cents.

Radix	2	2.5	2	2	2.5	2	2	2.5	2	10
Denomination	\$100	\$50	\$20	\$10	\$5	\$2	\$1	50c	20c	10c
Position (cents)	10000	50000	20000	1000	500	200	100	50	20	10

Mixed-radix systems do not usually provide any performance benefit and are often converted to a single-radix system for computation. Using a mixed-radix does not preclude redundant number systems since a redundant system only requires a symbol alphabet with cardinality greater than a radix. For example, your wallet is a redundant number system if there are multiple coins or notes of the same denomination. The value of 50c can be made up of a single 50c coin, or two 20c coins and a 10c coin, hence multiple representations of 50c.

2.4 HAMMING WEIGHT

The Hamming weight of a representation is the count of nonzero symbols. Some comparative examples of Hamming weights are given in Table 2.3. Increasing the radix of the representation decreases the number of symbols required for a particular value. However, the probability of a zero symbol also decreases. Alternatively, a redundant symbol alphabet can be chosen in order to increase the number of zero symbols, hence decrease the Hamming weight.

Table 2.3: Examples of the Hamming weight of various representations for a value of 317.

Common Name	Representation	Hamming Weight
Binary	100111101_2	6
Ternary	102202_3	4
Octal	475_8	3
Decimal	317_{10}	3
Hexadecimal	$13D_{16}$	3
Carry-Save	20022221_2	6
Binary Signed Digit	$101000\overline{1}01_2$	4

Let $W(Y)$ be the Hamming weight of the representation $Y = y_{\langle N-1\rangle}, \cdots, y_{\langle 0\rangle}$, thus,

$$W(Y) = \sum_{i=0}^{N-1} \delta(y_{\langle i\rangle}), \tag{2.6}$$

where,

$$\delta(y_{\langle i\rangle}) = \begin{cases} 0, & y_{\langle i\rangle} = 0, \\ 1, & y_{\langle i\rangle} \neq 0 \end{cases} . \tag{2.7}$$

For every value there exists a BSD representation that has a Hamming weight at most the same though usually less than that of binary. Both use the same radix of two. BSD achieves an increase in zero symbol frequency by replacing strings of 1-bits with a $\overline{1}$-symbol at the least significant 1-bit and a 1-symbol in the postion above the most significant 1-bit, all other 1-bits are replaced with a 0-symbol. For example, the binary word for 15 is 01111_2. This is replaced by $1000\overline{1}_2$ as a BSD word. This replacement becomes obvious when the representation is evaluated with (2.1); the binary representation value is calculated as $8 + 4 + 2 + 1 = 15$ and the BSD representation value as $16 + -1 = 15$.

A Hamming weight reduction is not always available for redundant number systems as demonstrated by the carry-save number system. It does not have a mechanism for replacing 1-bits with 0-symbols. The extra 2-symbol in its alphabet merely relocates a 1-bit right one position. For example, the binary word for 30 is 11110_2. This can be replaced by 02222_2 as a minimum Hamming weight carry-save word achieving no reduction in Hamming weight.

Hamming weight takes its name from Richard Wesley Hamming, an American mathematician who worked in the field of error correction codes among others. He

introduced Hamming weight in [Hamming, 1950]. The use of Hamming weight in coding theory and number systems is quite different. Coding theory seeks to *maximise* the minimum Hamming weight (or Hamming distance) as this improves a code's error correcting capability [Haykin, 2000]. Redundant number systems, however, try to *minimise* the average or maximum Hamming weight, since this can reduce the computational effort [Xu et al., 2007, Arno and Wheeler, 1993, Dempster and Macleod, 1995, Phillips and Burgess, 2004]. Care should be taken not to confuse these opposing targets in different technology domains.

Hamming weight is an important concept and is referred to often. Chapter 3 will consider it in greater detail. Sections 3.1 and 3.2 discuss methods for converting a non-redundant representation into a redundant representation of minimum Hamming weight. Section 3.3 uses a Markov analysis on the converting state machine to determine a number system's average minimum Hamming weight. In Chapter 10 the average minimum Hamming weight will be used as a criteria for selecting a symbol alphabet and radix for a number system. Some symbol alphabets will be analysed in Section 10.4.

2.4.1 Canonical Signed Digit

Canonical signed digit (CSD) is a subset of BSD. An extra condition is asserted; every nonzero symbol is separated by at least one zero symbol [Hwang, 1979, Koc, 1996]. Coincidentally, all CSD representations are minimum Hamming weight BSD representations. The extra restriction on CSD removes the redundancy present in BSD, leaving only a single valid representation for each value, though it still has a redundant alphabet. This makes it a pseudo-redundant number system.

A simple serial algorithm exists for converting from two's complement binary to CSD [Hwang, 1979]. It scans the binary word from least significant bit (LSB) to most significant bit (MSB), replacing all non-singular groups of 1-bits with a $1,\overline{1}$ CSD symbol pair delimiting a group of 0-symbols. For example, the sequence $0111_2 = 4+2+1 = 7_{10}$ is replaced by $1000\overline{1}_2 = 8-1 = 7_{10}$. Some further examples are shown in Table 2.4.

A systematic and implementable method of performing this recoding is as follows. Beginning from the LSB of the binary word,

1. sequentially scan the binary word in steps of one bit, looking at a window of two bits; the current bit $b_{\langle i \rangle}$ and the next bit $b_{\langle i+1 \rangle}$. Match one of the three conditions:
   - $b_{\langle i \rangle} = 0$; output a CSD 0-symbol; continue from 1,
   - $b_{\langle i+1 \rangle}, b_{\langle i \rangle} = 0, 1$; output a CSD 1-symbol; continue from 1,

Table 2.4: Examples of binary words recoded to canonical signed digit (CSD).

#	2's Complement	CSD	Decimal	Recoded	Explanation
1	0001_2	0001_2	1	No	Single 1 not recoded.
2	0011_2	$010\overline{1}_2$	3	Yes	Group of 2 recoded to satisfy CSD rules.
3	0101_2	0101_2	5	No	Two non-consecutive single 1s not recoded.
4	0111_2	$100\overline{1}_2$	7	Yes	Group of 3 recoded to satisfy CSD rules.
5	11100_2	$00\overline{1}00_2$	-4	Yes	The terminating 1 lying outside the width giving the correct negative result.
6	11011_2	$00\overline{1}0\overline{1}_2$	-5	Yes	Recoding of lower group of 2 as in #2 creates a group of 3 as in #5.

- $b_{\langle i+1\rangle}, b_{\langle i\rangle} = 1, 1$; output a CSD $\overline{1}$-symbol; continue from 2,

2. sequentially scan the binary word in steps of one bit, looking at a window of two bits; the current bit $b_{\langle i\rangle}$ and the next bit $b_{\langle i+1\rangle}$. Match one of the three conditions:

   - $b_{\langle i\rangle} = 1$; output a CSD 0-symbol; continue from 2,
   - $b_{\langle i+1\rangle}, b_{\langle i\rangle} = 1, 0$; output a CSD $\overline{1}$-symbol; continue from 2,
   - $b_{\langle i+1\rangle}, b_{\langle i\rangle} = 0, 0$; output a CSD 1-symbol; continue from 1,

This is equivalent to the Moore finite state machine shown in Figure 2.2.

The state machine transforms the binary word from LSB ($b_{\langle 0\rangle}$) to MSB ($b_{\langle N\rangle}$). For each step it looks at the current bit ($b_{\langle i\rangle}$) and the next bit ($b_{\langle i+1\rangle}$) to determine the transition to take. The binary input word is sign extended to keep the $b_{\langle i+1\rangle}$ input valid. The starting state is $S0$. The current state determines the output CSD symbol ($d_{\langle i\rangle}$), ignoring the first symbol of the starting state.

The two's complement binary words in examples #5 and #6 of Table 2.4 represent negative numbers. In these two cases the terminating CSD 1-symbol falls outside the word length so it is not placed. This is desirable behaviour as it gives the correct *negative* value for the resulting CSD representation. CSD is an implicitly signed representation as described in Section 2.2.If the binary input number is instead interpreted as unsigned then the CSD output word length should be extended by one and the terminating 1-symbol placed. Alternatively, this could be done by zero extending the input word.

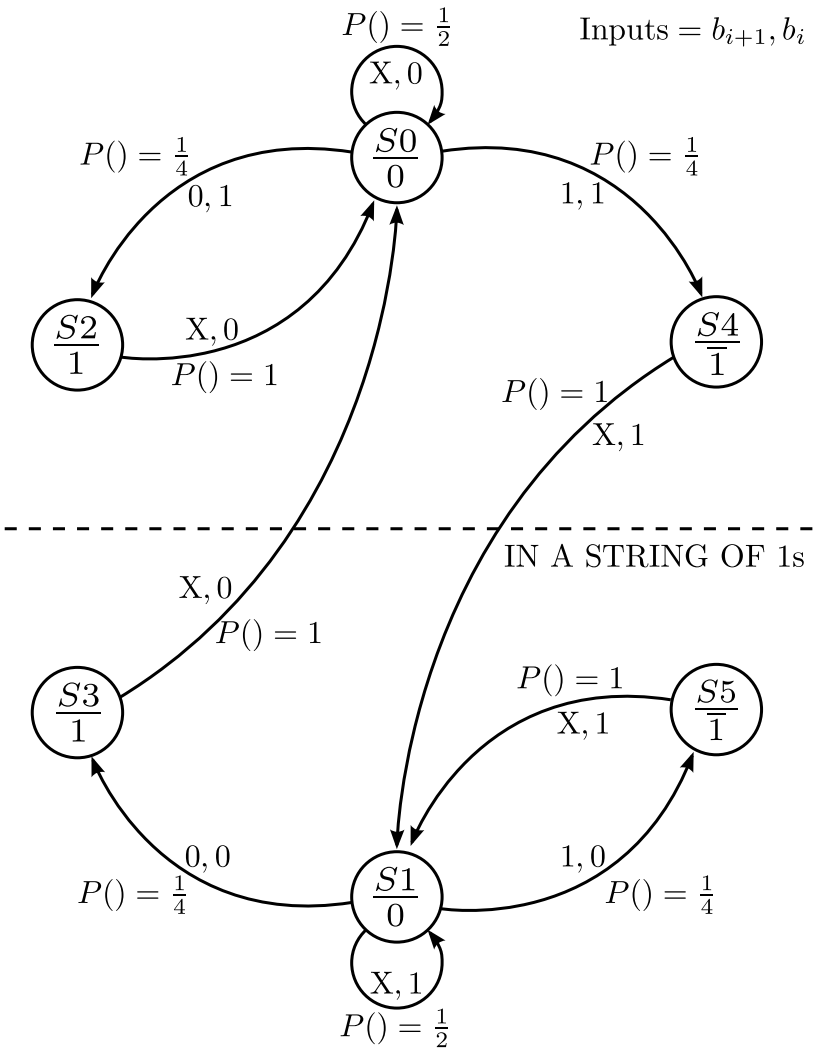


Figure 2.2: A finite state machine that serially converts two's complement binary into CSD. Within the transition conditions, the X means 'don't care'; either 1 or 0.

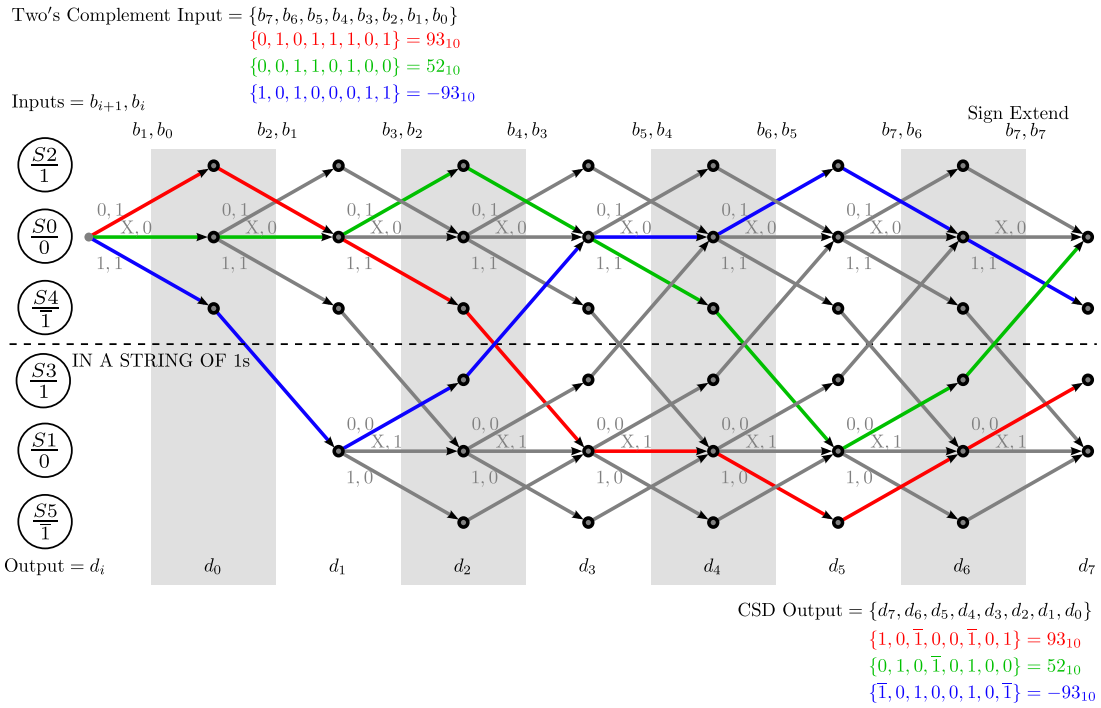


Figure 2.3: A trellis diagram showing the recoding of the binary words for 93, 52, and -93 into CSD.

Three examples of binary two's complement conversion are shown in Figure 2.3 as a trellis diagram to show the traversal of transitions and states.

2.5 RANGE

The range is the interval of integers that a number system can represent. The magnitude of the range is largely determined by the word length (N), however, the symbol alphabet affects which values can be represented at the extremes. Therefore, three ranges can be defined. In order of non-increasing size these are:

1. The *full* range encloses all representable values. However, there may be gaps in the representable numberline. The most negative value is represented by a repetitive string of the most negative symbol s_-, e.g., $s_- \cdots s_{-\,\beta}$. Similarly, the most positive value is represented by a repetitive string of the most positive symbol s_+, e.g., $s_+ \cdots s_{+\,\beta}$. This gives the *full* range,

$$R_{\text{full}} = \left[\sum_{i=0}^{N-1} s_- \beta^i, \quad \sum_{i=0}^{N-1} s_+ \beta^i\right], \tag{2.8}$$

$$= \left[s_- \frac{\beta^N - 1}{\beta - 1}, \quad s_+ \frac{\beta^N - 1}{\beta - 1}\right]. \tag{2.9}$$

2. The *continuous* range encloses the representable values that are consecutive and include zero. This range is contained within the full range and has limits at the first gap in the integer number line above and below zero. If $s_\downarrow$ is the most negative symbol that has consecutive symbols between it and zero, then the most negative value is represented by a repetitive string of $s_\downarrow$, e.g., $s_\downarrow \cdots s_{\downarrow\,\beta}$. Similarly, if $s_\uparrow$ is the most positive symbol that has consecutive symbols between it and zero, then the most positive value is represented by a repetitive string of $s_\uparrow$, e.g., $s_\uparrow \cdots s_{\uparrow\,\beta}$. This gives the *continuous* range,

$$R_{\text{cont}} = \left[\sum_{i=0}^{N-1} s_\downarrow \beta^i, \quad \sum_{i=0}^{N-1} s_\uparrow \beta^i\right], \tag{2.10}$$

$$= \left[s_\downarrow \frac{\beta^N - 1}{\beta - 1}, \quad s_\uparrow \frac{\beta^N - 1}{\beta - 1}\right]. \tag{2.11}$$

   For example, a number system with $\mathbb{S} = \{\overline{3}, \overline{1}, 0, 1, 2, 5\}$ would have $s_\downarrow = \overline{1}$ and $s_\uparrow = 2$. With $\beta = 3$ and $N = 4$, this gives the range $[-1 \times 40, 2 \times 40] = [-40, 80]$. The next most negative number outside the range is 41 and would need to be written either as $\overline{1112}_3$, but $\overline{2} \notin \mathbb{S}$, or as $\overline{1}0000_3$, but $N > 4$. Similarly, the next most positive number outside the range is 81 and would need to be written either as 2223_3, but $3 \notin \mathbb{S}$, or as 10000_3, but $N > 4$.

3. The *minimum weight* range encompasses a consecutive range of values with minimum Hamming weight. To be included, a value must first be representable with minimum Hamming weight with at most N symbols. If a value's minimum Hamming weight representation requires $N+1$ symbols it is not included. Secondly, the range is defined to be continuous about zero meaning the positive and negative boundaries stop at the first non-minimum Hamming weight value in their respective directions. If a value is outside these bounds it is not included.

   This range is useful when determining a scale factor that ensures the post-scaled values are representable with their minimum Hamming weight. This range is highly dependent on the redundancy created by the symbol alphabet. With a redundant alphabet it is difficult to define generally, although the positive and negative limits are usually greater than a factor $1/\beta$ of the limits of the continuous range, R_{cont}.

These ranges are all equivalent for non-redundant number systems where the symbol alphabet consists of consecutive integers, for example, decimal and binary. Figure 2.4 shows the ranges for the $N = 5$ symbol representations of the $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$, $\beta = 2$ number system.

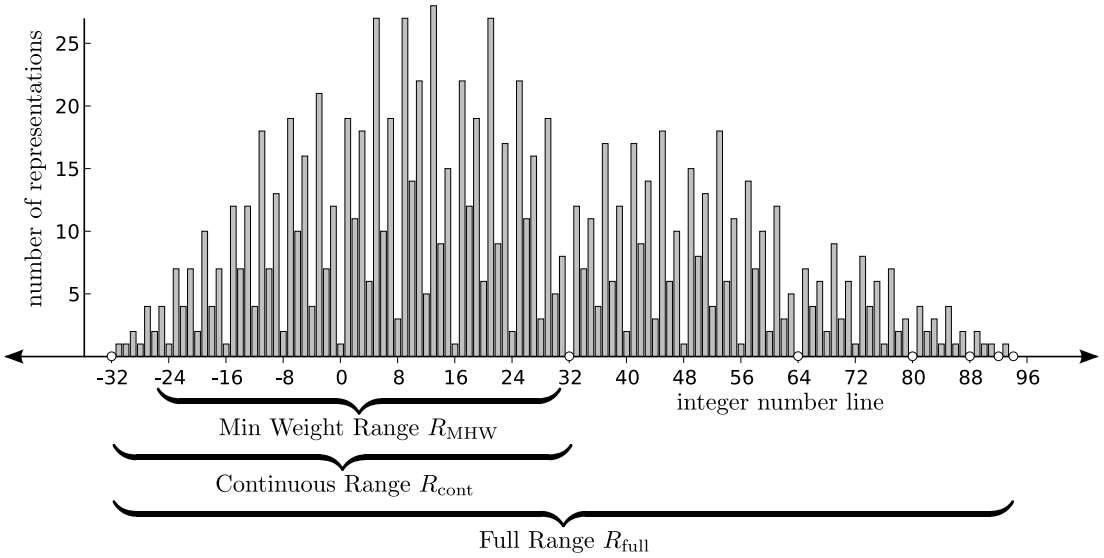


Figure 2.4: An integer numberline for the number system with $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$ and $\beta = 2$ and $N = 5$. The histogram shows the number of representations for each value. The circular points show which integers do not have representations. The three ranges are also shown below the number line.

2.6 NUMBER SYSTEM AND REPRESENTATION VISUALISATION

A number system can be visualised as a fully connected graph with the nodes corresponding to the symbols in the alphabet, as shown by the example in Figure 2.5. A number representation is a traversal of this graph, recording the visited symbol nodes.

Placing restrictions on which arcs can be traversed can be used to force the representations to have particular properties. For example, removing all arcs that do not enter or leave the node for the 0-symbol forces representations to have at least one zero symbol between every nonzero symbol. Doing this for the BSD number system gives the canonical signed digit (CSD) number system which happens to have unique and minimum Hamming weight representations for each value [Xu et al., 2007, Hwang, 1979].

A number system can also be visualised as an N-dimensional space, where each dimension corresponds to a position in a representation. Each representation is a unique point in that space, where the value along the ith dimension is the value of the symbol $s_{\langle i \rangle}$. This representation space is flattened onto the one dimensional number line by (2.1). Multiple points in the representation space may map to the same point on the number line, meaning it is a redundant number system. An arithmetic operation on

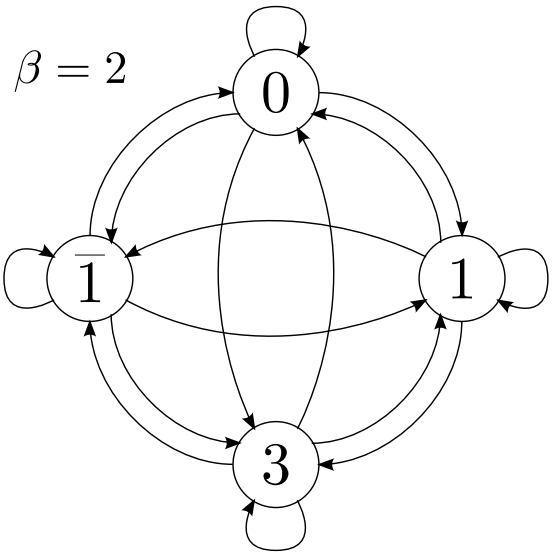


Figure 2.5: A number system graph for the number system with $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$ and $\beta = 2$.

a representation will move from one point in the space to another. Reversing this operation to find the representations of the operands would usually not be possible, even if you know the results representation and the operands values.

2.7 TREE ALGORITHM TO GENERATE ALL REPRESENTATIONS OF A VALUE

A representation value graph or tree can be created where the nodes are the representations' values as calculated by (2.1). Each value can have a large number of representations as estimated by (2.5). Finding all representations for a value v is not a trivial task. A naive approach is to enumerate all $|\mathbb{S}|^N$ representations and store those that evaluate to v using (2.1). An algorithm is developed below that will find all the representations of a value v given an arbitrary radix β and symbol alphabet $\mathbb{S}$.

The algorithm generates a representation tree that starts with the value $V = v$ at the root node. At each branch a symbol is appended to the representation, starting at the LSS and appending towards the MSS. The node at the end of a branch has the value of its parent less the symbol, i.e., the value remaining to be encoded. The algorithm is applied recursively to the nodes, terminating when a node reaches zero remaining value. This indicates a finite length representation has been found.

The symbol alphabet $\mathbb{S}$ can be defined as the union of β disjoint subsets,

$$\mathbb{S} = \bigcup_{\gamma=0}^{|\beta|-1} \mathbb{S}_\gamma, \tag{2.12}$$

$$\mathbb{S}_\gamma = \{s_i : s_i \mod \beta = \gamma\}. \tag{2.13}$$

For example, the number system with $\mathbb{S} = \{\overline{1}, 0, 2, 3\}$ and $\beta = 2$ has subsets $\mathbb{S}_0 = \{0, 2\}$ and $\mathbb{S}_1 = \{\overline{1}, 3\}$.

From (2.1) the value V can be split into two additive parts, the value V_l remaining to be encoded at position l, and that already encoded up to position $l-1$,

$$V = V_l\beta^l + \sum_{n=0}^{l-1} s_{\langle n\rangle}\beta^n. \tag{2.14}$$

The integer value V_l that has $V_l \mod \beta = \gamma$ can only release a symbol s_i from the set $\mathbb{S}_\gamma$. This gives a new smaller value to be encoded at position $l+1$ and a new symbol at position l,

$$V_l\beta^l = V_{l+1}\beta^{l+1} + s_{i,\langle l\rangle}\beta^l. \tag{2.15}$$

For example, given a value $V_0 = 15$ to encode in BSD with alphabet $\mathbb{S} = \{\overline{1}, 0, 1\}$ and radix $\beta = 2$. The two subsets of $\mathbb{S}$ are $\mathbb{S}_0 = \{0\}$ and $\mathbb{S}_1 = \{\overline{1}, 1\}$. Since $\gamma = 15 \bmod 2 = 1$, the two symbols that can be used in the LSS position are from $\mathbb{S}_1 = \{\overline{1}, 1\}$. Two branches are created, one for each of $s = 1$ and $s = \overline{1}$. Releasing the symbol $s = \overline{1}$ gives, $15 = 8 \times 2^1 + \overline{1} \times 2^0$, where removing $\overline{1}$ leaves 16 to be encoded with $V_1 = 8$ at the child node.

Rearranging and simplifying gives a simple recursive equation for the value at the next node given the symbol released,

$$V_{l+1} = (V_l - s_{i,\langle l\rangle})/\beta. \tag{2.16}$$

Note that the division by β has zero remainder, since s_i is from $\mathbb{S}_\gamma$. From this a recursive algorithm can be defined for building a representation tree. One branch for each s_i in $\mathbb{S}_\gamma$ is created from the parent node with value V_l at level l. The child node has the value defined by (2.16). A representation can be read by traversing the tree and noting the symbols assigned to the branches. The LSS is the branch from the root node, and the MSS is a branch to a leaf node with remaining value $V_l = 0$.

Drawing the representation tree as a graph, by combining nodes with the same value V_l, exposes cases for infinite recursion as cycles in the graph, see Figure 2.6 as an example. These cases necessitate a termination of the recursion when the desired word

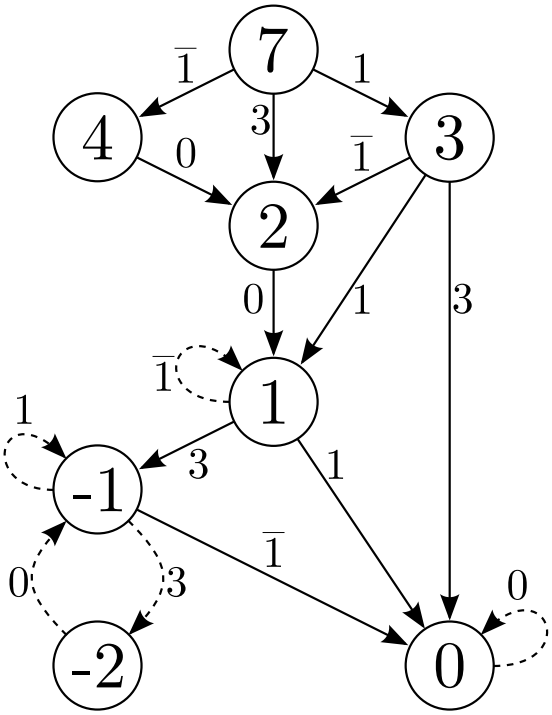


Figure 2.6: The representation graph of the value 7 using the number system $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$. The nodes are labelled with the remaining value to be coded (V_l) and the edges with the symbol (s_i) to be added to the MSS of the representation. Four infinitely recursive cycles are present; shown as dotted lines.

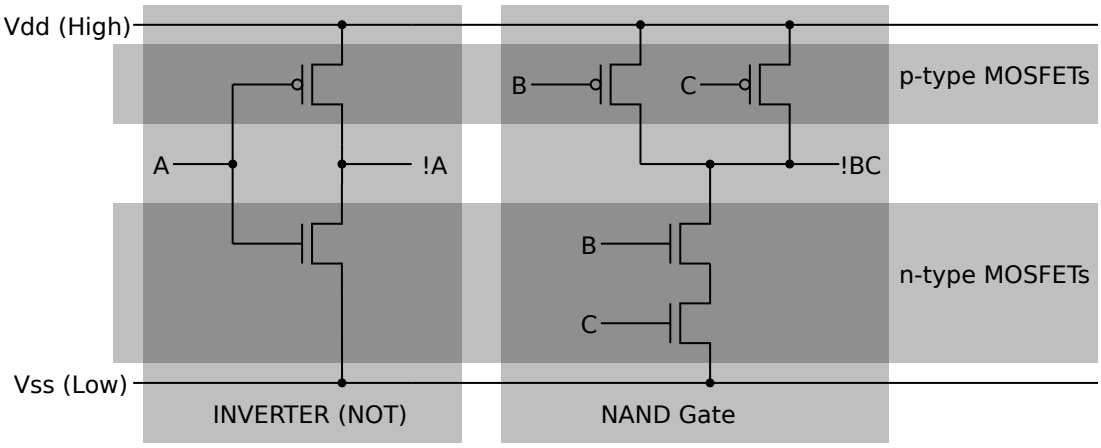


Figure 2.7: Digital circuits for complementary metal-oxide-semiconductor (CMOS) inverter and NAND gates.

length (N) has been reached. At termination some leaf nodes will be left with nonzero values; these nodes and corresponding branches should be discarded.

This algorithm generalises the one presented in [Dempster and Macleod, 2004] for BSD by branching on an arbitrary symbol alphabet and any nonzero integer radix. Of course, these algorithms can only find representable values that adhere to the fixed radix system of (2.1).

2.8 MAPPING TO DIGITAL LOGIC

Current computer systems are ususally built using digital logic on a complementary metal-oxide-semiconductor (CMOS) integrated circuit (IC) [Kang and Leblebici, 2002]. The digital gates in CMOS circuits consist of a complimentary pair of metal-oxide-semiconductor field effect transistors (MOSFET), a p-type and an n-type arranged as shown in Figure 2.7. When the input of the inverter is driven to a high voltage the n-type MOSFET switches to an on state providing a low resistance path for current to flow from the output to ground. At the same time the p-type MOSFET switches to an off state providing a high resistance, limiting the current from the positive supply to the output. The output has a high resistance so this causes the output voltage to be pulled low. Similarly a low input voltage turns the n-type MOSFET off and the p-type MOSFET on, driving the output to a high voltage. Critically, signals in digital circuits have two stable states, a high voltage or a low voltage. These are usually labelled H and L, or 1 and 0, or True and False respectively. Some work has been done on ternary CMOS circuits, but they are much larger and slower [Gundersen et al., 2005].

To encode more than two states, several digital signals are grouped together. For

Table 2.5: Two common mappings between digital signals and the BSD symbol alphabet $\{\overline{1}, 0, 1\}$ with duplicate mappings for the zero symbol.

Symbol	Positive-Negative (p,n)	Sign-Magnitude (s,m)
$\overline{1}$	(0,1)	(1,1)
0 and	(0,0)	(0,0)
0	(1,1)	(1,0)
1	(1,0)	(0,1)

example, two digital signals have four states, $\{LL, LH, HL, HH\}$. The number of digital signals, D, required to uniquely encode each symbol in a symbol alphabet $\mathbb{S}$ is,

$$D = \lceil \log_2(|\mathbb{S}|) \rceil, \tag{2.17}$$

and D digital signals provide 2^D states.

BSD has one of the smallest symbol alphabets that gives a redundant number system, with only three symbols ($\mathbb{S} = \{\overline{1}, 0, 1\}$). To encode the three symbols requires the combined states of two digital signals. This encoding of symbols is repeated at every position along the representation. Therefore, a number system with a large symbol alphabet requires more logic resources to transmit, store, and perform operations on its representations.

If the number of digital states is greater than the alphabet cardinality ($D > |\mathbb{S}|$), such as with BSD, there are several options for dealing with the unused digital states.

1. Ignore the unused states. This may allow the logic to be optimised, saving some logic resources. The transmission and storage resources will not be reduced.

2. Assign the unused states to some of the symbols, doubling them up. This is commonly used for BSD where the three symbols $\{\overline{1}, 0, 1\}$ are mapped onto two digital signals as shown in Table 2.5. The zero symbol is mapped twice. This may again reduce the logic resources, since the logic may not need to do a complete transform on the representation signals.

3. Increase the size of the symbol alphabet such that $|\mathbb{S}| = D$. This can increase the number system redundancy and may result in other benefits such as reduced Hamming weight if the additional symbols are chosen well.

To implement a redundant number system efficiently in digital logic the number of signals should be minimised, i.e., limited to two digital signals per symbol; $|\mathbb{S}| \leq 4$. This

has a flow-on effect of reducing the combinational logic required to perform operations such as addition.

2.9 NUMBER SYSTEMS OF INTEREST

The number of number system specifications is boundless. The cardinality of the symbol alphabet is limited by the practical size of the implementing logic, but a cardinality of 64 symbols is possible while only using six digital signals per position. The choice of symbols is the next consideration, of which there are virtually unlimited choices of value and combinations of symbols. Finally, the choice of a radix can be any value, usually an integer with a magnitude greater than one. It may even be a complex number or irrational, such as the golden ratio ($\varphi \approx 1.61803$) [Bergman, 1957].

The work presented in the rest of this thesis is kept general, avoiding restrictions wherever possible to allow the exploration of the number system space. Some specific designs and most examples are illustrated with the use of radix-2 number systems with an alphabet of four or less symbols. The symbols themselves are usually small in magnitude. These choices are made mostly for practical reasons that are explained and reconsidered in Chapter 10. The zero symbol is particularly important and is the subject of Chapter 3. The symbol alphabet will always include the zero-symbol. Of course this is not an absolute requirement, but many advantages are derived from its inclusion.

Chapter 3

ZERO "FREEBIE! ZIP! ZERO! NADA!"[1]

Zero is arguably the most important number. It has two unique properties that make it extremely useful when performing arithmetic:

1. Zero is the additive identity. That is to say, adding zero to a number is equal to the number itself,
$$x + 0 = 0 + x = x \tag{3.1}$$
Similarly for subtraction,
$$x - 0 = x, \text{ and} \tag{3.2}$$
$$0 - x = -x. \tag{3.3}$$

2. Zero is the multiplicative absorbing element. That is any number multiplied by zero is equal to zero,
$$x \times 0 = 0 \times x = 0. \tag{3.4}$$
Similarly for division, zero divided by any number, other than zero, is equal to zero,
$$0/x = 0. \tag{3.5}$$
However, any number divided by zero, including $0/0$, is undefined [Suppes, 1999].

With both of these properties, if a zero is known to be one operand then the answer is trivial. It does not require any calculation, hence, no time or resources. This is exploited as often as possible in hardware designs. For example, to calculate a multiplication N partial products are generated and accumulated. The partial products are formed as the product of the multiplicand and a symbol of the multiplier operand. From (3.4) it is apparent that partial products with zero value are generated by zero-symbols from the

[1] "Freebie! Zip! Zero! Nada!", is an expression used on the TV show Monster Garage (2002-2006) when the build team gets something for nothing. This is appropriate for this chapter, since if a number is zero then operations with it are essentially free. See `http://en.wikipedia.org/wiki/Monster_Garage`.

multiplier representation. Hence, from (3.1), these zero partial products do not need to be accumulated. Therefore, for each zero symbol in the multiplier representation, neither the partial product generation nor the addition needs to be performed by the arithmetic hardware. This is explained in more detail in Section 7.1.

One strategy to minimise the time and resources spent on calculations is to maximise the instances of zero-symbols in the representations, i.e., choose the representations with minimum Hamming weight. Of all a value's redundant representations only a few will have minimum Hamming weight. Section 3.1 summarises some prior work in generating minimum Hamming weight representations for a specified value. Unfortunately, this prior algorithm is restricted in the number systems that it can encode. Therefore, to allow the investigation of irregular number systems, a new method is developed in Section 3.2 to overcome the prior's limitations. This results in a Mealy finite state machine. In Section 3.3 the finite state machine is subjected to a Markov analysis, revealing the individual symbol probabilities at each position in the word. This is used in Section 3.4 to determine a linear expression for the average minimum Hamming weight of a number system.

3.1 REGULAR SLIDING WINDOW MINIMUM HAMMING WEIGHT SYMBOL SET CONVERSION

To reduce the calculations in multiplication-like algorithms, first the minimum Hamming weight representations must be found. The common method is to serially recode the non-redundant representation of a value, for example, binary. These serial algorithms query a finite window of symbols and replace them with a sequence of symbols from the new redundant alphabet, predominately the zero symbol. It is essentially a pattern replacement problem.

A sliding window algorithm for performing minimum Hamming weight digit set conversion is described in [Phillips and Burgess, 2004]. Their method employs a window of m symbols $(x_{\langle j+m-1\rangle} \cdots x_{\langle j\rangle})$ that slides up a non-redundant representation $(x_{\langle N-1\rangle} \cdots x_{\langle i\rangle} \cdots x_{\langle 0\rangle})$ from least significant symbol (LSS) to most significant symbol (MSS). A second variable c_i is kept to represent a carry and has a value in the set $\{0, 1\}$ and initially zero. This gives the window sum,

$$w_i = c_i + \sum_{j=0}^{m-1} x_{i+j}\beta^j, \tag{3.6}$$

Whenever the window sum modulo β is nonzero, $x_{\langle i\rangle}$ is replaced by the appropriate redundant symbol $y_{\langle i\rangle}$, the selection of which is given later. If the redundant symbol

is negative then the carry variable, c, is set to 1, otherwise it is set to 0. The $m-1$ symbols following the redundant symbol are set to zero.

This method only works for redundant symbol alphabets satisfying,

$$\mathbb{S} = \{s : l \leq s \leq u, s \bmod \beta \neq 0\} \bigcup \{0\}, \tag{3.7}$$

where the lower limit l is,

$$-\beta^m < l \leq 0, \quad l \in \mathbb{Z}, \tag{3.8}$$

and the upper limit u is,

$$1 \leq u < \beta^m, \quad u \in \mathbb{Z}, \tag{3.9}$$

and finally, the minimum cardinality of the alphabet is $|\mathbb{S}| \geq \beta^m - \beta^{m-1} + 1$. Together, these conditions mean that there must be at least one symbol to represent every β^m possible windowed selections of symbols in the input non-redundant representation. Table 3.1 gives the possible symbol alphabets for number systems with $\beta = 2$ that can be encoded by this method.

Table 3.1: Radix-2 symbol alphabets that can be minimum weight encoded with the generalised sliding window method of Phillips and Burgess [2004].

m	$\mathbb{S}$	l	u
1	$\{0, 1\}$	0	1
2	$\{\overline{1}, 0, 1\}$	-1	1
2	$\{0, 1, 3\}$	0	3
2	$\{\overline{1}, 0, 1, 3\}$	-1	3
2	$\{\overline{3}, \overline{1}, 0, 1\}$	-3	1
3	$\{0, 1, 3, 5, 7\}$	0	7
3	$\{\overline{1}, 0, 1, 3, 5\}$	-1	5
3	$\{\overline{3}, \overline{1}, 0, 1, 3\}$	-3	3
3	$\{\overline{5}, \overline{3}, \overline{1}, 0, 1\}$	-5	1

The output symbol $y_{\langle i \rangle}$ is chosen conditionally on the value of w_i from (3.6), the carry variable c_i, and the upper and lower output symbol alphabet values u and l respectively. The output symbol is given by,

$$y_i = \begin{cases} w_i - \beta^m, & w_i > u \\ w_i - \beta^m, & w_i - c_i = \beta - 1 \text{ and } w_i \geq l + \beta^m \\ w_i, & \text{otherwise} \end{cases} . \tag{3.10}$$

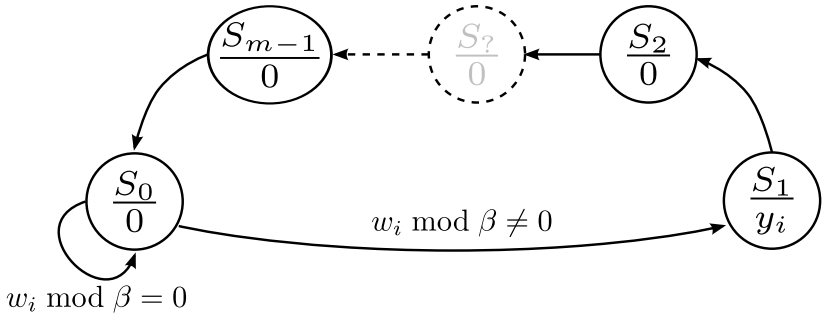


Figure 3.1: Moore state diagram of the generalised sliding window minimum Hamming weight digit set algorithm. The dotted state symbolises $m-2$ repeated states that output a zero symbol.

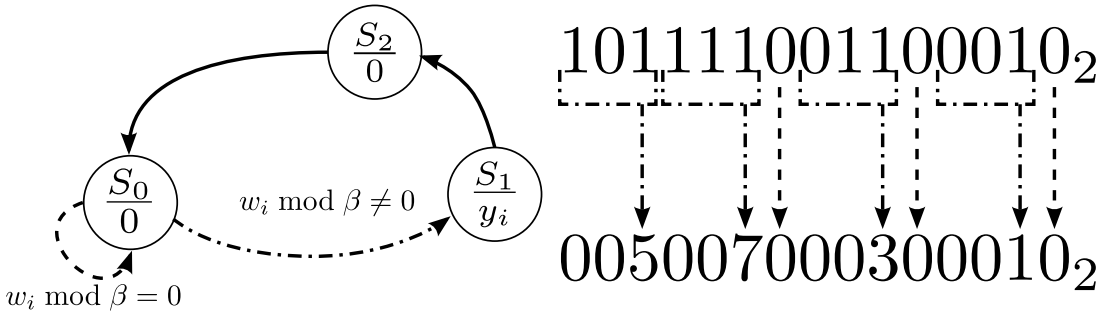


Figure 3.2: An example of recoding a binary representation to a redundant alphabet ($\mathbb{S} = \{0, 1, 3, 5, 7\}$) to achieve a reduced Hamming weight using the regular sliding window approach with window width of $m = 3$ and radix $\beta = 2$.

This algorithm is described by the Moore state machine of Figure 3.1. At each state a new symbol is emitted to the minimum weight representation. The algorithm begins in state 0 where it spins outputting 0-symbols while $x_{\langle i \rangle} + c \bmod \beta = 0$. The transition to state 1 is taken when $x_{\langle i \rangle} + c \bmod \beta \neq 0$. The output symbol for state 1 is chosen using (3.10). This output symbol covers the windowed value, so states 2 through $m-1$ output 0-symbols. For example, the binary representation 1011110011010_2 is recoded, as shown in Figure 3.2, with the alphabet $\mathbb{S} = \{0, 1, 3, 5, 7\}$ to give 0050070003010_2 with minimum Hamming weight.

3.2 IRREGULAR ALPHABET MINIMUM HAMMING WEIGHT CONVERSION.

The earlier technique of Phillips and Burgess [2004] presented in Section 3.1 is limited in the legal encoding alphabets, in that they must satisfy a number of rules. These alphabets all must have consecutively valued symbols, ignoring those that are a multiple of the

radix. For example, the number system $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$, $\beta = 2$ is valid, where symbol-2 is ignored because it is a multiple of β. However, the number system $\mathbb{S} = \{\overline{1}, 0, 1, 5\}$,$\beta = 2$ cannot be encoded since the algorithm wrongly assumes there is a 3-symbol.

To overcome this limitation and to allow other more exotic number systems to be investigated in this thesis, a new sliding window algorithm was developed to handle irregularly spaced symbols. The new algorithm places no limitations on the continuity of alphabet symbols and supports minimum Hamming weight encoding of irregular number systems such as $\mathbb{S} = \{\overline{1}, 0, 1, 5\}$, $\mathbb{S} = \{\overline{3}, 0, 1, 7\}$, and $\mathbb{S} = \{0, 1, 5, 7, 11\}$.

The new algorithm consists of three parts:

1. Find all minimum Hamming weight recodings for a relatively small range of values and place them in a recoding table.
2. Reduce the recoding table to have the smallest recoding window size to create a mapping table.
3. Build a Mealy state machine that serially implements the mapping table, extending the recoding past the window width.

The linking mechanism between the parts are tables associating a windowed section of the non-redundant input word to its minimum Hamming weight replacement, more specifically the LSS of the replacement. Part 1 uses a branch and bound tree algorithm to find the minimum Hamming weight representations and is presented in Section 3.2.1. Part 2 groups the possible minimum Hamming weight representations, reducing the recoding table to a minimum number of entries and is presented in Section 3.2.2. Part 3 is presented in Section 3.2.3, where the mapping table is used to build a minimum Hamming weight recoding Mealy state machine.

3.2.1 Minimum Hamming weight recoding table

The recoding table lists all the minimum Hamming weight representations for values in the range $[0, \beta^M)$. These are the values of a windowed section of the input word, $x_{\langle i+M-1\rangle}, \cdots, x_{\langle i\rangle}$, of length M symbols. M is the largest acceptable window width and sets the maximum search space. M needs to be selected such that it is large enough to find a consistent minimum Hamming weight encoding. The window width used in the final state machine is m and it is found in Section 3.2.2. Therefore, M should satisfy $m \leq M \leq N$ (the word-length), which may require trial and error to determine, or become too computationally expensive to explore as M gets large.

The recoding table entries consist of the windowed value and a list of minimum Hamming weight representations, which may have any width. Only the LSS $s_{\langle i \rangle}$ will be used for the state machine. The rest of the minimum Hamming weight representation will be inferred by the state machine later. The replacement strings are generated for a redundant number system defined by symbol alphabet $\mathbb{S}_r$ and radix β. To efficiently find all the minimum Hamming weight representations for each value $v \in [0, \beta^M)$, a branch and bound technique is employed. A tree such as Figure 3.3 is built by the algorithm. Each leaf node represents a windowed value. The transitions' outputs along each path back to the root node give a minimum Hamming weight representation and the inputs give the windowed string. The tree is built breadth-wise, branching to a new level of nodes as the window width is increased. Only one level of nodes is maintained in order to reduce the memory footprint of the algorithm. The previous level of nodes are deleted, as in a work-list style algorithm. Several attributes are stored by the node data structure of Listing 3.1. These are derived from the Mealy transitions' input and output symbols along the path from the root node to itself. The stored attributes are:

1. The input string's value along a path of length l from the root node, $\sum_{i=0}^{l-1} x_{\langle i \rangle} \beta^i$. This must be identical for all paths to the node.
2. The propagate value, the difference between the input string and the encoded string, projected to the next tree level, $\left[\sum_{i=0}^{l-1} \left(x_{\langle i \rangle} - s_{\langle i \rangle}\right) \beta^i\right] / \beta$. This must be identical for all paths to the node.
3. The Hamming weight of the encoded string along all paths from the root node. This must be identical for all paths and is used as the node merging or removal criteria.
4. A set of the LSS for all paths leading to this node, i.e., the symbol(s) output on each path's transition from the root node.

During the algorithm some nodes are merged, so there may be multiple paths giving rise to multiple LSS. This is desirable behaviour as it may allow the final state machine to be smaller.

The tree begins from a root node with the default (zeroed) attributes at level $l = 0$. From each node the input symbol $x_{\langle i+l \rangle}$ can take one of β different values. Each input symbol could result in several non-redundant symbols to be emitted. For each permutation of input x and output symbol s a transition to a new node is created if and only if,

$$(x + p) \bmod \beta \equiv s \bmod \beta, \quad x \in [0, \beta), \quad s \in \mathbb{S}_r, \tag{3.11}$$

where p is the propagate attribute of the node. The transition has an input condition of $x_{\langle i+l \rangle} = x$ and a Mealy output of $s_{\langle i+l \rangle} = s$.

Listing 3.1: Definition of a data structure for storing node attributes for the FindEncodingTable algorithm.

```
1: class node()
2:     self.value ← 0                          ▷ Value of input symbol string.
3:     self.propagate ← 0 ▷ Difference between value and output symbol string value.
4:     self.weight ← 0                  ▷ Hamming weight of output symbol string(s).
5:     self.lss ← {}                    ▷ Set of LSS from paths leading to this node.
6: end class
```

Each transition terminates at a new node in the next level. The node's value and propagate attributes are derived from the transition's input and output symbols as previously described. If two nodes have equivalent value and propagate attributes then the node with the largest Hamming weight attribute is deleted along with its transitions. If the two nodes' Hamming weights are equivalent then the nodes are merged with the union of their LSS sets and incoming transitions. By this mechanism, only the minimum Hamming weight encoding paths are kept in the tree and paths with greater Hamming weight are pruned as soon as a better one is found.

When the number of levels equals the maximum window width $l = M$, $P = \lceil \log_\beta(\max(\mathbb{S})) \rceil + 1$ more levels are generated with $x = 0$ as the input symbol in (3.11). This allows the propagate values to converge to zero and as they do so the nodes will merge or eliminate each other until one leaf node for each value remains. The algorithm thus far is presented in Listing 3.2 as pseudo code. An example of a completed search tree for BSD and $M = 4$ is shown in Figure 3.3.

Upon completion of the tree there will be β^M leaf nodes, one for each maximum-window value and all with propagate attributes equal to zero. The recoding table summarises the tree, listing the window values against a set of LSSs. A set may contain multiple symbols if more than one path back to the root node exists. For example, the BSD number system generates Table 3.2, where values 3, 11, and 13 have multiple paths and hence two symbols in their LSS sets, $\{\overline{1}, 1\}$.

Notice in Table 3.2 that for each value $v \bmod \beta = 0$ (the even values when $\beta = 2$) the output symbol is zero. This allows an optimisation in the tree generation where the root node does not need to generate a transition for $x_{\langle i \rangle} = 0, s_{\langle i \rangle} = 0$.

3.2.2 Minimum window width

The next stage of the algorithm attempts to find the minimum window width m that will give a minimum Hamming weight encoding. This will make the finite state machine smaller and if implemented in hardware, faster and more resource efficient. Beginning

Listing 3.2: Algorithm to find the least significant symbol(s) of the minimum weight representations of the numbers $[0, \beta^M)$ for the number system with alphabet $\mathbb{S}$, and radix β.

```
 1: procedure FindEncodingTable(𝕊, β)
 2:     P ← {new node}                ▷ Initialise parent node list with default root node
 3:     for l = 0, l < M, l = l + 1 do
 4:         P ← MakeMinWeightEncodingLevel(𝕏, 𝕊, β, P, l)
 5:     end for
 6:     for l = M, l < M + P, l = l + 1 do
 7:         P ← MakeMinWeightEncodingLevel({0}, 𝕊, β, P, l)
 8:     end for
 9:     return P
10: end procedure
11:
12: procedure MakeMinWeightEncodingLevel(𝕏, 𝕊, β, P, l)
13:     C ← {}                                          ▷ Initialise empty child list
14:     for each p ∈ P and x ∈ X and s ∈ S do
15:         if s − x − p.propagate mod β = 0 then
16:             c ← new node()
17:             if p.lss = {} then
18:                 c.lss ← {s}
19:             else
20:                 c.lss ← p.lss
21:             end if
22:             c.value ← p.value + x × β^l
23:             c.propagate ← (x + p.propagate − s)/β
24:             c.weight ← p.weight
25:             if s ≠ 0 then
26:                 c.weight ← c.weight +1
27:             end if
28:             d ← node ∈ C similar to c else None
29:                                   ▷ similar means same value & propagate.
30:             if d ≠ None then
31:                 if d.weight > c.weight then
32:                     C ← C − {d}                          ▷ Remove d from C
33:                     C ← C ∩ {c}                          ▷ Add c to C
34:                 else if d.weight = c.weight then
35:                     c.lss ← c.lss ∩ d.lss                ▷ Merge c & d
36:                     C ← C ∩ {c}                          ▷ Add c to C
37:                 end if
38:             else
39:                 C ← C ∩ {c}                              ▷ Add c to C
40:             end if
41:         end if
42:     end for
43:     return C
44: end procedure
```

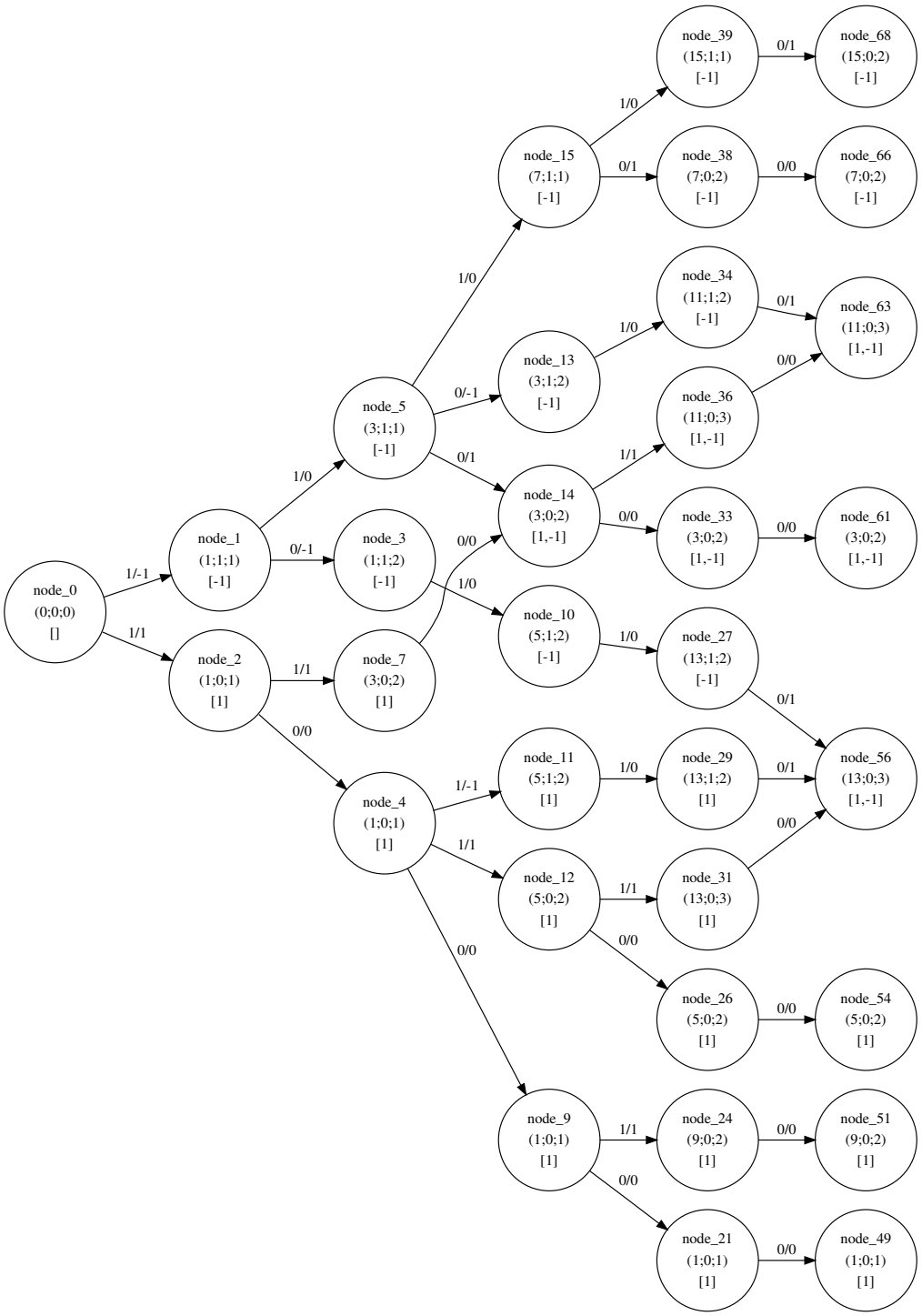


Figure 3.3: The minimum Hamming weight encoding tree for BSD ($\mathbb{S} = \{-1, 0, 1\}$, $\beta = 2$) with $M = 4$ as generated by the algorithm of Listing 3.2. The nodes are labelled as "node_ID(value; propagate; weight)[LSS set]". The arcs are labelled input symbol / output symbol. The branch from node_0, the root node, with input symbol 0 and output symbol 0 has been ignored.

Table 3.2: Recoding table for binary to minimum Hamming weight BSD. Showing windows of maximum length $M = 4$ and the LSS sets that will give a minimum weight encoding.

v	$x_{M-1}, \cdots, x_0$	$\longrightarrow$	$s_{\langle 0 \rangle}$ set
0	0_2	$\longrightarrow$	$\{\ 0\ \}$
1	1_2	$\longrightarrow$	$\{\ 1\}$
2	10_2	$\longrightarrow$	$\{\ 0\ \}$
3	11_2	$\longrightarrow$	$\{\overline{1}\ 1\}$
4	100_2	$\longrightarrow$	$\{\ 0\ \}$
5	101_2	$\longrightarrow$	$\{\ 1\}$
6	110_2	$\longrightarrow$	$\{\ 0\ \}$
7	111_2	$\longrightarrow$	$\{\overline{1}\ \}$
8	1000_2	$\longrightarrow$	$\{\ 0\ \}$
9	1001_2	$\longrightarrow$	$\{\ 1\}$
10	1010_2	$\longrightarrow$	$\{\ 0\ \}$
11	1011_2	$\longrightarrow$	$\{\overline{1}\ 1\}$
12	1100_2	$\longrightarrow$	$\{\ 0\ \}$
13	1101_2	$\longrightarrow$	$\{\overline{1}\ 1\}$
14	1110_2	$\longrightarrow$	$\{\ 0\ \}$
15	1111_2	$\longrightarrow$	$\{\overline{1}\ \}$

with a window width of $m = 1$, the table entries are grouped into β^m sets according to their value v modulo β^m,

$$G_k = \{v : v \bmod \beta^m = k, 0 \leq v < \beta^M\}. \tag{3.12}$$

The LSS sets of each value within a group are then compared, looking for a contradiction – one value that does not have a set that coincides with the others. That is, the intersection of the LSS sets of values in G_k is the empty set. For the BSD example, with a window width of $m = 1$, the $\beta^m = 2$ groups from Table 3.2 are:

- $G_0 = \{0, 2, 4, 6, 8, 10, 12, 14\}$ ($v \bmod 2^1 = 0$) and,
- $G_1 = \{1, 3, 5, 7, 9, 11, 13, 15\}$ ($v \bmod 2^1 = 1$).

In G_0 all the values agree that the 0-symbol should be the LSS as seen in Table 3.2. However, in G_1 the LSS sets for values 1 and 7 from Table 3.2 contradict each other. The value 1 only wants a 1-symbol output, while the value 7 only wants a $\overline{1}$-symbol output to generate their minimum Hamming weight representations. The value 3 does not contradict either value 1 or 7 since it can have either a 1-symbol or a $\overline{1}$ symbol as

its output. There are no contradictions in G_0 since all its values want the 0-symbol as the output.

If a contradiction is found then the window width is increased and the LSS sets are checked again. In the BSD example, for a window width of $m = 2$, the $\beta^m = 4$ groups are:

- $G_0 = \{0, 4, 8, 12\}$ ($v \bmod 2^2 = 0$),
- $G_1 = \{1, 5, 9, 13\}$ ($v \bmod 2^2 = 1$),
- $G_2 = \{2, 6, 10, 14\}$ ($v \bmod 2^2 = 2$) and,
- $G_3 = \{3, 7, 11, 15\}$ ($v \bmod 2^2 = 3$).

There are no contradictions within these groups and so the minimum width of $m = 2$ and the mapping of Table 3.3 is obtained. Pseudo code to implement this process is presented in Listing 3.3.

Table 3.3: minimum Hamming weight mapping table for binary to minimum Hamming weight BSD with a window width of $m = 2$.

G_x	x_{i+1}, x_i	values	$\longrightarrow$	$s_{\langle i \rangle}$
G_0	$0, 0$	$\{0, 4, 8, 12, \ldots\}$	$\longrightarrow$	0
G_1	$0, 1$	$\{1, 5, 9, 13, \ldots\}$	$\longrightarrow$	1
G_2	$1, 0$	$\{2, 6, 10, 14, \ldots\}$	$\longrightarrow$	0
G_3	$1, 1$	$\{3, 7, 11, 15, \ldots\}$	$\longrightarrow$	$\overline{1}$

Having the minimum window width will give the smallest recoding state machine in Section 3.2.3. However, a larger window width may give flexability in choosing a particular minimum Hamming weight recoding. For example, in BSD the value 3 can be minimum Hamming weight encoded as $10\overline{1}_2$ or 011_2. If the 011_2 representation is desired then a window width of $m = 3$ is required to differentiate it from 7 and 15 that still need $\overline{1}$ as their LSS. With $m = 3$ this gives $G_3 = \{3, 11\}$ a least significant symbol set of $\{\overline{1}, 1\}$ meaning that either choice will give a minimum Hamming weight encoding. Manually restricting the least significant set for G_3 to be $\{1\}$ in Table 3.2 will ultimately give the 011_2 encoding for 3.

3.2.3 Minimum Hamming weight recoding state machine

Using the recoding table, a state machine can be constructed to recode a long ($N \gg m$) non-redundant representation into a redundant representation with minimum Hamming

Listing 3.3: Algorithm to take the list P with values 0 to $\beta^M - 1$ associated with their LSS sets for encoding to the symbol alphabet $\mathbb{S}$ and radix β. The algorithm reduces the list to a new mapping with values 0 to β^m where $m < M$. The new mapping has minimum m such that values v and $v + \beta^m$ require the same LSS for encoding for all $v < \beta^{M-1}$.

```
1: procedure FindWindowedValueToLSSMapping(M, P, 𝕊, β)
2:     for m ∈ [1, M) do
3:         mapping ← {}                ▷ A set of mappings for windowed value to LSS(s).
4:         contradiction ← False                ▷ Flag variable to break out of loops.
5:         for v ∈ [0, β^m) do
6:             set ← 𝕊        ▷ Initialising the set of possible LSSs for the mapping to v.
7:             for each node ∈ P do
8:                 if node.value mod β^M = v and node.propagate = 0 then
9:                     set ← set ∩ node.lss  ▷ Remove symbols from set not in node.lss.
10:                    if set = {} then          ▷ No elements of node.lss remain in set.
11:                        contradiction ← True
12:                        break
13:                    end if
14:                    mapping ← mapping ∪ {v ⇒ set}
15:                end if
16:            end for
17:            if contradiction = True then
18:                break                                  ▷ Try new window width.
19:            end if
20:        end for
21:        if contradiction = False then          ▷ Found a self-consistent mapping.
22:            return m, mapping
23:        end if
24:    end for
25:    return M, {}                                ▷ No consistent mapping found.
26: end procedure
```

weight. The process outlined in this section constructs a Mealy state machine with transitions conditioned on the MSS of the window. Each transition shifts the window along the non-redundant word by one symbol, and a single redundant symbol is output. This construction means that there are β transitions out of each state, each with an equal probability of $1/\beta$. Of course, transforms can be applied to the state machine to reduce the number of states at the expense of an increase in the complexity of choosing a transition, i.e., looking at a greater number of symbols in the window. This is avoided as it breaks the Markov assumption employed in Section 3.3, making the analysis of the state machine unnecessarily difficult.

The input condition of the Mealy transitions is an m length window of the non-redundant representation, $x_{\langle i+m-1\rangle} \cdots x_{\langle i\rangle\ \beta}$. The window consists of a decision symbol, $x_{\langle i+m-1\rangle}$, and a base condition, $b_k = x_{\langle i+m-2\rangle} \cdots x_{\langle i\rangle\ \beta}$. The base condition is common to all transitions into and out of a state k. The base condition is really a property of the state, but for clarity it is also attributed to the transitions.

The output symbol $s_{\langle i\rangle}$ for a transition is taken from the minimum Hamming weight mapping table T, as generated in Section 3.2.2, at the entry value t,

$$t = (x_{\langle k+m-1\rangle}\beta^m + b_k + p_k) \bmod \beta^m; \tag{3.13}$$

$$s_{\langle i\rangle} \leftarrow T(t), \tag{3.14}$$

where, p_k is an attribute of the state. Each state has a unique pairing of attributes:

1. A value b_k in the range $(0, \beta^{m-1}]$. This is the value of the $m-1$ length base condition. From an incoming transition's condition it can be calculated as,

$$b_k = \sum_{j=0}^{m-2} x_{\langle i+j\rangle}. \tag{3.15}$$

For an outgoing transition, the next state's base condition can be calculated as,

$$b_{k+1} = \sum_{j=1}^{m-1} x_{\langle i+j\rangle}. \tag{3.16}$$

2. A propagate value, being the difference of the input and output strings to that point,

$$p_k = \sum_{j=0}^{i} \left(x_{\langle j\rangle} - s_{\langle j\rangle}\right) \tag{3.17}$$

The propagate value is negative if the output string has borrowed some value from future input windows and positive if the output string's value lags the input and

needs to be made up in future transitions. To avoid remembering all past inputs and outputs, the next state's propagate, p_{k+1}, is calculated from the current state's propagate, p_k, and the connecting transition's input and output symbols, $x_{\langle i \rangle}$ and $s_{\langle i \rangle}$ respectively,

$$p_{k+1} = \lceil p_k/\beta \rceil + \lfloor (x_{\langle i \rangle} - s_{\langle i \rangle})/\beta \rfloor. \tag{3.18}$$

The attribute pair (b_{k+1}, p_{k+1}) uniquely identifies a state and can be determined from the incoming transitions' attributes and their outgoing state's propagate attribute.

The state machine is built recursively starting from the reset state S_{R}, where the propagate value $p_{\langle 0 \rangle} = 0$. The transitions out of the reset state are special, as they do not have a base condition. The full window of m input symbols is used to determine the next state. Therefore, β^m transitions are created from the reset state, one for each of the mapping table entries. A transition's end state is identified by calculating the propagate value p_{k+1} from (3.18) and the base condition b_{k+1} from (3.16). If the state (b_{k+1}, p_{k+1}) does not exist then it is created with β new transitions out to be resolved. The state machine is complete when all transitions' end states are resolved. The minimum Hamming weight encoding Mealy state machine for BSD with a window width of $m = 2$ is shown in Figure 3.4. A larger encoding state machine is shown in Figure 3.5 for the number system $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$.

3.3 MARKOV ANALYSIS OF MINIMUM HAMMING WEIGHT ENCODING STATE MACHINES TO DETERMINE SYMBOL PROBABILITIES

A Markov chain is a sequence of random variables, or states, $X_1, X_2, X_3, \ldots$ with the Markov property; a future state only depends on the current state and no previous states [Bhat, 2008]. The minimum Hamming weight encoding state machines are designed with this in mind. Having the Markov property means that transitions have a stationary probability of being selected. In the case of the minimum Hamming weight encoding state machines, the transition probabilities are $1/\beta$, with the exception of those from the reset state which have a probability of $1/\beta^m$.

To perform the Markov analysis the encoding state machine is first translated into a matrix description as follows:

1. Create a state transition probability matrix $\mathbf{T}$, where each column corresponds to the start state and each row corresponds to the end state.

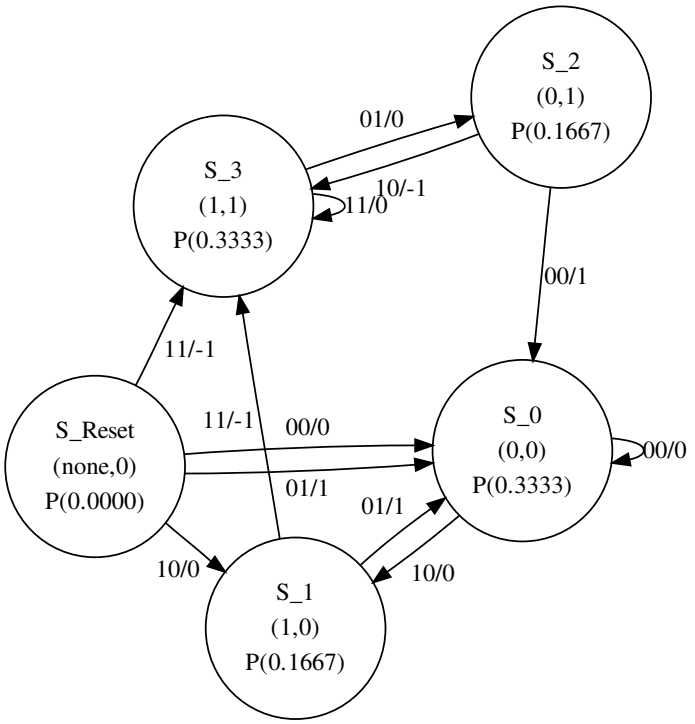


Figure 3.4: The minimum Hamming weight encoding Mealy state machine for BSD with the minimum window width of $m = 2$. The states are labelled with a name (S_x), their unique attribute pair (b_k, p_k), and the probability of being in that state as $i \to \infty$. The transitions are labelled as input condition/output symbol, i.e., $x_{\langle i+m-1 \rangle} \cdots x_{\langle i \rangle} / s_{\langle i \rangle}$.

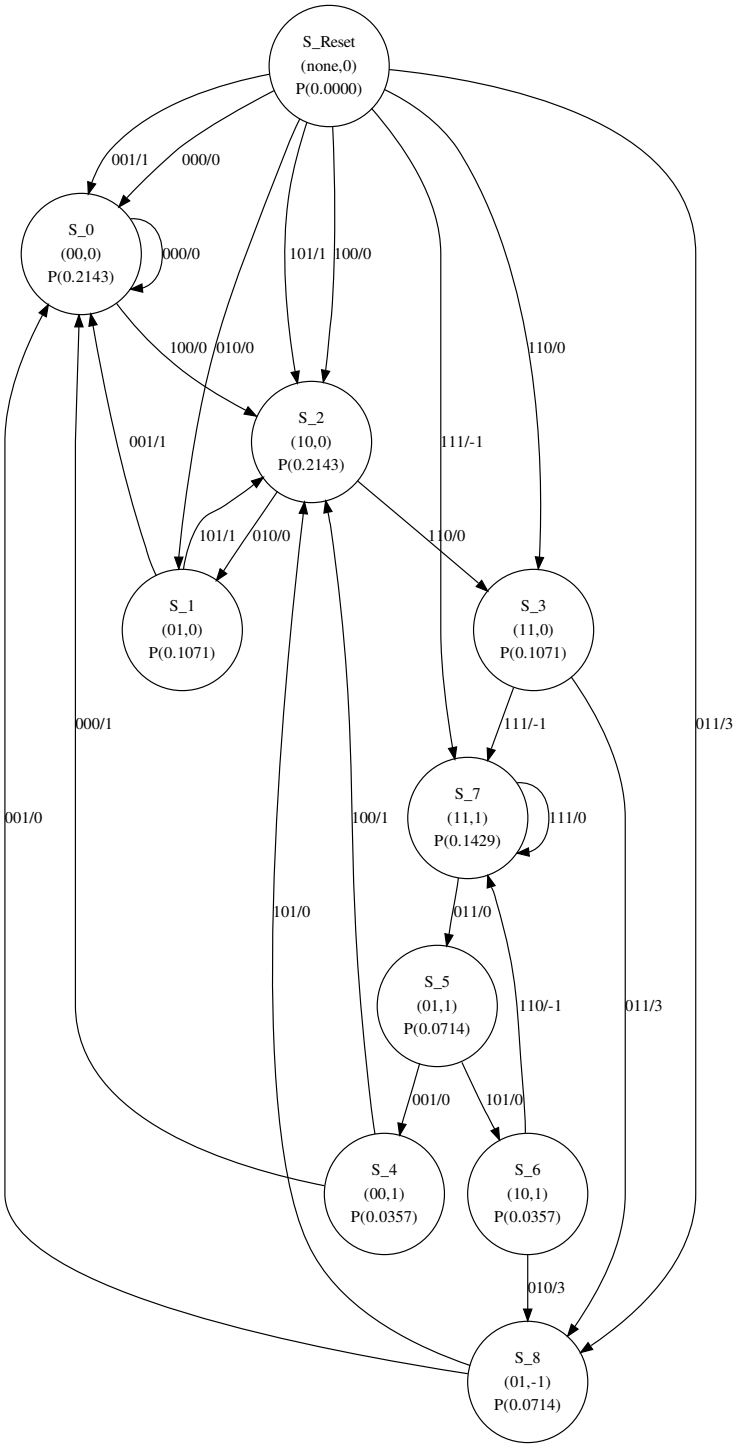


Figure 3.5: The minimum Hamming weight encoding Mealy state machine for the number system $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$ with the minimum window width of $m = 3$. The states are labelled with a name S_x, their unique attribute pair (b_k, p_k), and the probability of being in that state as $i \to \infty$. The transitions are labelled as input condition/output symbol, i.e., $x_{\langle i+m-1 \rangle} \cdots x_{\langle i \rangle} / s_{\langle i \rangle}$.

2. Populate $\mathbf{T}$ with the transition probabilities, where $\mathbf{T}_{\langle i,j \rangle} = 1/\beta$ for the transition from start state j to end state i. Transitions from the reset state will have probabilities that are multiples of $1/\beta^m$. All other entries are 0 since those transitions do not exist.

3. Initialise a state probability column vector $\mathbf{P}_0$ where each element corresponds to a state as in the transition matrix. Set the element corresponding to the reset state to 1 and all others to 0 since, with probability 1 the state machine starts in the reset state.

For the BSD example, from Figure 3.4 there are five states labelled S_Reset, S_0, S_1, S_2, and S_3. The transition matrix can be constructed from the transitions between these states as,

$$\mathbf{T} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{4} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{4} & 0 & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}. \tag{3.19}$$

Each column of $\mathbf{T}$ should sum to a probability of 1 since there will always be a valid transition from the corresponding state. The state machine always starts in the reset state, therefore, the initial state probability vector is,

$$\mathbf{P}_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{3.20}$$

The state probabilities at the next step, $\mathbf{P}_{k+1}$, can be calculated from the current state probabilities, $\mathbf{P}_k$, and the transition matrix according to the recursive relationship,

$$\mathbf{P}_{k+1} = \mathbf{T}\mathbf{P}_k. \tag{3.21}$$

The elements of $\mathbf{P}_k$ should always sum to 1 since no transition terminates the state machine. To find the state probabilities at the kth step, (3.21) is recursively substituted into itself to give,

$$\mathbf{P}_k = \mathbf{T}^k\mathbf{P}_0. \tag{3.22}$$

As $k \to \infty$ the state probabilities converge to a steady-state solution $\mathbf{P}_\infty$. By using

(3.21) in the limit, the steady-state solution can be posed as an eigenvector problem,

$$\lambda \mathbf{P}_\infty = \mathbf{T}\mathbf{P}_\infty. \tag{3.23}$$

The steady-state solution is the eigenvector of $\mathbf{T}$ corresponding to the eigenvalue of $\lambda = 1$ [Poole, 2006]. Usually, the eigenvector is normalised to a Euclidean norm of 1, however, to interpret the result as a probability the eigenvector must be rescaled such that its elements sum to 1. For the BSD example, this gives the steady-state solution,

$$\mathbf{P}_\infty = \begin{bmatrix} 0 \\ \frac{1}{3} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{3} \end{bmatrix}. \tag{3.24}$$

Finally, to translate the state probabilities into symbol probabilities, a third matrix $\mathbf{S}$ is constructed to represent the probability of outputting a symbol given a state. There is one column for each symbol in $\mathbb{S}$ and one row for each state. Like the transition matrix it is constructed by inspecting the transitions from each state and the symbols output. For example, from Figure 3.4, the symbol output matrix is,

$$\mathbf{S} = \begin{bmatrix} \frac{1}{4} & \frac{2}{4} & \frac{1}{4} \\ 0 & \frac{2}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & \frac{2}{2} & 0 \end{bmatrix}, \tag{3.25}$$

where the columns correspond to the symbols $\overline{1}$, 0, and 1 respectively, and the rows correspond to the states as in $\mathbf{P}$. The symbol probabilities at position i in the recoded word can then be calculated as,

$$\Pr(\mathbb{S}, i) = \mathbf{S}^T \mathbf{T}^i \mathbf{P}_0. \tag{3.26}$$

The BSD example gives the progression,

$$\begin{aligned}
\Pr(\mathbb{S}, 0) &= \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}, \\
\Pr(\mathbb{S}, 1) &= \begin{bmatrix} \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \end{bmatrix}, \\
\Pr(\mathbb{S}, 2) &= \begin{bmatrix} \frac{3}{16} & \frac{5}{8} & \frac{3}{16} \end{bmatrix}, \\
\Pr(\mathbb{S}, 3) &= \begin{bmatrix} \frac{5}{32} & \frac{11}{16} & \frac{5}{32} \end{bmatrix}, \\
\vdots & \quad \vdots \\
\Pr(\mathbb{S}, \infty) &= \begin{bmatrix} \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}.
\end{aligned} \tag{3.27}$$

At high word positions the symbol probabilities converge to $\mathbf{S}^T\mathbf{P}_\infty$, which for the BSD example is $\begin{bmatrix} \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}$.

3.4 MINIMUM HAMMING WEIGHT EXPECTATION

A useful metric to compare number systems is the expected number of nonzero-symbols in a representation that is minimum Hamming weight encoded. In this section the expected minimum Hamming weight is calculated in a closed form that assumes an infinite word length, hence an infinite range of values. The result is derived from the symbol alphabet and the radix. This is opposed to the average minimum Hamming weight where the weight of the minimally encoded representations of all values in a range are averaged. The average minimum Hamming weight is an estimate of the expected minimum Hamming weight. The literature, however, does not usually make this distinction and average minimum Hamming weight is the more often used term [Xu et al., 2007, Khabbazian et al., 2005, Wu and Hasan, 1999] and therefore used throughout this thesis.

The expected minimum Hamming weight is,

$$H(V) = E\{W_{\min}(V)\}, V \sim \mathrm{U}(R_{\min}), \tag{3.28}$$

where $W_{\min}(V)$ is the function returning the minimum Hamming weight of V, which has a uniform distribution over the minimum Hamming weight range.

The average minimum Hamming weight is usually expressed as a fractional factor of the word length N and is calculated assuming an infinite word length, $N \to \infty$ [Koc and Hung, 1992, Wu and Hasan, 1999, Phillips and Burgess, 2004]. This factor can be interpreted as the probability of the infinite position having a nonzero-symbol,

$$\widehat{H} \approx 1 - \Pr(s_{\langle\infty\rangle} = 0). \tag{3.29}$$

In the BSD example of Section 3.3, the probability of outputting a zero symbol is $\Pr(s_{\langle\infty\rangle} = 0) = 2/3$. Therefore, BSD has a fractional factor of 1/3; in the long run a third of all symbols are nonzero. In reality, the probability of a nonzero-symbol converges quickly, making it a good estimate for small word lengths as well.

There is also a small, often ignored [Koc and Hung, 1992, Phillips and Burgess, 2004], constant contribution to the average minimum Hamming weight that arises because the Markov process is not started from the steady-state distribution. This can be calculated as the sum of differences between $\Pr(\mathbb{S}, i)$ and $\Pr(\mathbb{S}, \infty)$,

$$\mathbf{C}_N = \sum_{i=0}^{N-1} \left(\mathbf{S}^T \mathbf{T}^i \mathbf{P}_0 - \mathbf{S}^T \mathbf{T}^\infty \mathbf{P}_0 \right), \tag{3.30}$$

$$\mathbf{C}_N = \sum_{i=0}^{N-1} \mathbf{S}^T \left(\mathbf{T}^i - \mathbf{T}^\infty \right) \mathbf{P}_0, \tag{3.31}$$

$$\tag{3.32}$$

For the BSD example, the constant contribution vector, $\mathbf{C}_N$, with an element for each symbol, converges at large N to,

$$\mathbf{C}_N = \begin{bmatrix} \frac{1}{18} & -\frac{1}{9} & \frac{1}{18} \end{bmatrix}. \tag{3.33}$$

Summing elements corresponding to the nonzero symbols gives a constant factor of 1/9. This should be offset by the zero symbol element, to give $\mathbf{C}_\infty$ a sum of zero. Therefore, the average minimum Hamming weight for BSD is,

$$\widehat{H_{\mathrm{BSD}}} = \frac{N}{3} + \frac{1}{9}. \tag{3.34}$$

This result is confirmed by the analytic results of Hartley [1996], Arno and Wheeler [1993], and [Wu and Hasan, 1999] for signed digit number systems. The method presented in this chapter will calculate the average minimum Hamming weight for any radix and symbol alphabet. Further number systems are analysed and their average minimum Hamming weights are computed in Chapter 10.

Chapter 4

ZERO DOMINANT REPRESENTATIONS

It was found in the optimisation of the dot-product, to be presented in Chapter 8, that minimising the Hamming weight of representations did not in all situations give optimal performance. Section 4.1 introduces zero-dominance as a novel property of redundant number representations where the positions of zero symbols are as important as minimising the Hamming weight. Representations posessing zero-dominance are collected in the zero-dominant set as described in Section 4.2. A dot-product calculation is used as the motivating application in Section 4.3 and investigated further in Chapter 8. An algorithm for generating the zero-dominant set for a specified value and redundant number system is presented in Sections 4.4 and 4.5.

4.1 ZERO-DOMINANCE

The zero-symbol is assumed to be the most desirable symbol since it is the additive identity element and multiplicative absorbing element, see Chapter 3. With these two properties, if an operand is known to be zero then the answer is trivial. This is why minimising the Hamming weight (maximising the zero-symbol count) is desirable. Zero-dominance extends this premise to give significance to the positions of zero-symbols.

Borrowing the concept of Pareto-optimality [Das and Dennis, 1998] from multi-objective combinatorial optimisation, the redundant representations of a value can be compared for Pareto-optimality. In a particular representation, a symbol position is given a score of 1 if the symbol is nonzero and a score of 0 if the symbol is zero. To be a Pareto-optimal representation means there is no other representation where the score of a position can be reduced without increasing the score of any other position. Representation **A** is said to zero-dominate (Pareto-improve) representation **B** when both:

1. **A** has a zero-symbol in every position that **B** has a zero-symbol and;

Table 4.1: Dominance relations for some $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$,$\beta = 2$ representations of the value seven.

A	relation	**B**
$\overline{1}311_2$	is a substitute for	$1\overline{1}11_2$
0111_2	dominates	$1\overline{1}11_2$
$100\overline{1}_2$	dominates	$1\overline{1}11_2$
$100\overline{1}_2$	is a complement of	0111_2
0031_2	dominates	0111_2
0031_2	is a complement of	$100\overline{1}_2$
0103_2	dominates	0111_2
0103_2	is a complement of	$100\overline{1}_2$
0103_2	is a complement of	0031_2

2. **A** has a lower cumulative score than **B**.

These conditions mean that each zero-dominant representation has at least one zero-symbol in a position that any other zero-dominant representation has a nonzero symbol.

Two situations arise when there is no dominance in either direction. If condition 1 does not hold, irrespective of condition 2, then **A** and **B** are complements; they provide alternate placements of zero symbols. If condition 1 holds but condition 2 does not, then **A** and **B** are substitutes; they can be exchanged without changing the zero positions.

Some examples of dominance relationships are given in Table 4.1. The examples used in this and following sections will use the number system $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$,$\beta = 2$ since it provides more interesting and illustrative patterns than BSD ($\mathbb{S} = \{\overline{1}, 0, 1\}$,$\beta = 2$). The additional 3-symbol, with value of 3, provides a lower average minimum Hamming weight than BSD. This can be illustrated by considering that both the BSD patterns of 011 and $10\overline{1}$ with Hamming weights of 2 can be rewritten as 003 with a reduced Hamming weight of one.

4.2 ZERO-DOMINANT SET OF REPRESENTATIONS

The zero-dominant set (ZDS) is the set of all representations that are not dominated, i.e, the set of Pareto-optimal representations. They form a Pareto-frontier covering all other representaitons [Das and Dennis, 1998]. The example dominance relations of Table 4.1 results in the ZDS of $\{100\overline{1}_2, 0031_2, 0103_2\}$ for the value seven. The surviving representations have unique zero placements with all superfluous nonzero-symbols removed. For example, $1\overline{1}11_2$ is removed in favour of $100\overline{1}_2$.

All minimum Hamming weight representations are included in the ZDS, but not all zero-dominant representations are necessarily minimum weight. For example, the value 23_{10} has a ZDS of $\{0300\overline{1}_2, 10031_2, 10103_2\}$ with $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$. This contains representations of both Hamming weight 2 and 3. The ZDS also includes zero-dominant substitute representations. If uniqueness of zero-dominant representations with the same zero positions is required i.e., no substitutes, then one may be chosen on an arbitrary basis and others removed.

4.3 A ZDS APPLICATION

The ZDS is useful in the optimisation of dot-products with constant coefficients, for example, digital filters. The multiply accumulate unit hardware design presented in Section 9.2 and published in [Kamp and Bainbridge-Smith, 2007] packs the redundant number system coefficients together. For each shift in the multiplier the nonzero partial product can be selected from any one of the coefficients. This means that a zero partial product from one coefficient can be replaced with a non-zero partial product from another coefficient. This reduces the number of clock cycles required by computing different parts of the dot-product where a zero partial product would have wasted time.

The positions of the zero-partial product shifts can be manipulated by selecting different representations for the coefficients from their respective ZDS. The purpose is to make space for other coefficients' nonzero partial products, so that the shift with the largest number of nonzero partial products is minimised. It will be shown in Chapter 8 that for the dot-product of two L length vectors $\mathbf{A}$ and $\mathbf{B}$, optimal packing can reduce the effective number of multiplication cycles from L to approximately $L \times H_{\min}/N$, where $H_{\min}$ is the expected minimum Hamming weight of the coefficients in $\mathbf{A}$.

Consider a simple example with two coefficients that have the same value, as illustrated in Figure 4.1. Coefficients are often duplicated in real valued FIR filters.

1. If the values are generated by a Hamming weight minimising linear recoding [Phillips and Burgess, 2004], then both coefficients will have the same representation. With nonzero symbols in the same place no packing is achieved, as shown by the stacked dots in Figure 4.1(a). This is because the recoding is a one-to-one mapping.

2. If the minimum Hamming weight set of representations is used, then some packing may be achieved, as shown by fewer stacked dots in Figure 4.1(b).

3. If the representations are chosen from the ZDS then better packing can be achieved,

$343_{10} =$	1	0	1	0	0	3	0	0	$\overline{1}_2$
$343_{10} =$	1	0	1	0	0	3	0	0	$\overline{1}_2$
place	•		•			•			•
usage	•		•			•			•

(a) Simple linear minimum Hamming weight recoding.

$343_{10} =$	1	0	1	0	0	3	0	0	$\overline{1}_2$
$343_{10} =$	0	3	0	$\overline{1}$	0	$\overline{1}$	0	0	$\overline{1}_2$
place						•			•
usage	•	•	•	•		•			•

(b) Representations from the minimum Hamming weight set.

$343_{10} =$	1	0	0	3	$\overline{1}$	0	1	0	3_2
$343_{10} =$	0	3	$\overline{1}$	0	0	3	0	0	$\overline{1}_2$
place									•
usage	•	•	•	•	•	•	•		•

(c) Representations from the zero-dominant set.

Figure 4.1: Examples of packing of two equal valued coefficients using their redundant representations selected from different sets.

even with an increase in the total number of nonzero partial products, as shown by all but one column having single dots in Figure 4.1(c).

Now consider extending this to a larger selection of coefficients with each coefficient choosing a representation that fits with the others. A new algorithm to achieve this is described in Section 8.6.

The ZDS is the smallest set that contains all possible alternatives for zero locations. In comparison to the minimum Hamming weight representations, the ZDS provides extra capacity for better value packing. In the example of Figure 4.1, the ZDS has a representation that, although higher in weight, has a zero in the $s_{\langle 3 \rangle}$ place allowing the two representations to avoid having nonzero symbols in the same positions, except the unavoidable $s_{\langle 0 \rangle}$ place, see Section 8.7.1.

4.4 ZERO DOMINANT SET GENERATION ALGORITHM

To find a value's ZDS, all its representations must be generated and checked against each other for dominance. However, the number of representations (M) generated for each value is large, of the order $|\mathbb{S}|^N/\beta^N$, where $|\mathbb{S}|^N$ is the number of permutations for N symbols chosen from the symbol alphabet $\mathbb{S}$, and β^N is the approximate size of the representable integer range with N symbols. Hence dominance checking, an $O(M^2)$ test, is expensive. This cost can be reduced by employing a branch and bound technique [Lawler and Wood, 1966]. By checking for dominance while generating the representation tree, the nodes that lead to non-dominant representations can be removed early.

The representation tree is built breadth-wise, the same as the algorithm to generate all representations presented in Section 2.7. After each level is generated, the dominance is checked between nodes with the same value. If a node is dominated then it is removed from the tree, and so its children are never generated. The two subtrees generated below the dominating and dominated nodes, will be identical because they start with the same value. Therefore, only the paths to the two subtree roots needs to be examined for dominance.

The algorithm is presented as pseudo-code in Listing 4.1. It uses a work-list that stores the nodes in the current level of the representation tree. This is initialised with the root node (line 2). On the next iteration the parent nodes are replaced by their children that survive dominance checking (line 36). The algorithm terminates when all child nodes have zero value (line 3). The zero-dominant representations can then be extracted from the leaf nodes (line 38) constituting the ZDS.

4.5 AVOIDANCE OF CYCLIC CONDITIONS

The representations occurring from the infinitely recursive situation in Section 2.7 cannot form dominant representations. They will always be dominated by a similar representation that took a direct path to zero value avoiding the cyclic paths. While dominance checking alone will eventually remove these paths, the algorithm would still require a maximum word length termination condition. To alleviate this, the conditional instantiation of new nodes is included in lines 8, 10, 12, 19, and 21 of the algorithm to eliminate the cyclic paths. This also speeds up the algorithm and reduces memory requirements.

The cyclic paths exist in the domain of the symbol set. Figure 4.2 shows examples

Listing 4.1: Algorithm to generate the ZDS for value V, alphabet $\mathbb{S}$, and radix β.

```
1: procedure BUILDZDS(V, 𝕊, β)
2:     P ← {node(none, V, none)}                          ▷ Initialise parent node list
3:     while any p ∈ P has p.value ≠ 0 do                 ▷ Current nodes not zero
4:         C ← {}                                         ▷ Empty child list
5:         for each p ∈ P do
6:             V_p ← p.value                              ▷ Parent node's value
7:             if V_p ∈ 𝕊 then
8:                 if V_p = 0 then                        ▷ Zero extend parent's representation
9:                     C ← C + {node(0, 0, p)}            ▷ Concatenate list
10:                else if V_p ∈ 𝕊_0 then                 ▷ Special case to preserve dominance
11:                    C ← C + {node(V_p, 0, p), node(0, V_p/β, p)}   ▷ Concatenate list
12:                else                                   ▷ Shortcut path to zero node
13:                    C ← C + {node(V_p, 0, p)}          ▷ Concatenate list
14:                end if
15:            else                                       ▷ V_p ∉ 𝕊
16:                γ = V_p mod β
17:                for each s_i ∈ 𝕊_γ do
18:                    V_c ← (V_p − s_i)/β                ▷ Child node's value
19:                    if V_c ≠ V_p then                  ▷ Standard case
20:                        C ← C + {node(s_i, V_c, p)}    ▷ Concatenate list
21:                    else                               ▷ V_c = V_p
22:                        do nothing                     ▷ Don't traverse single arc cycles
23:                    end if
24:                end for
25:            end if
26:        end for
27:        for each c_1 ∈ C do                            ▷ Check for dominance between node pairs
28:            for each c_2 ∈ C do
29:                if c_1 dominates c_2 then
30:                    delete c_2
31:                else if c_2 dominates c_1 then
32:                    delete c_1
33:                end if
34:            end for
35:        end for
36:        P ← C                                          ▷ Next generation
37:    end while
38:    ZDS ← P                                            ▷ Result
39: end procedure
40:
41: class node(symbol, value, parent)                     ▷ Data structure for storing node details
42:     self.value ← value
43:     self.parent ← parent
44:     self.representation ← {symbol} + parent.representation
45: end class
```

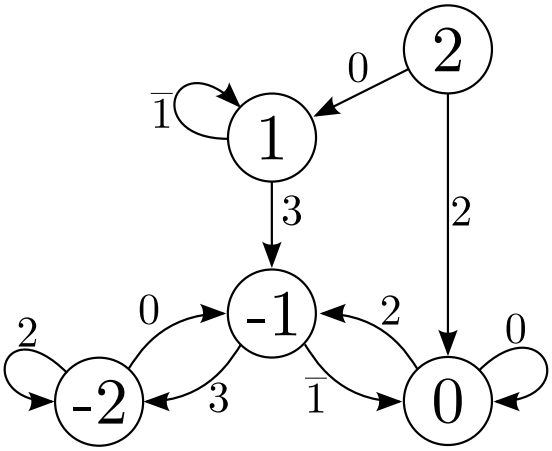


Figure 4.2: Examples of the recursive cycles in the algorithm using the number system $\mathbb{S} = \{\overline{1}, 0, 2, 3\}$, $\beta = 2$. The nodes are labelled with the remaining value to be encoded (V_l) and the edges with the symbol (s_i) to be added to the most significant symbol of the representation.

of the cases covered in the algorithm of Listing 4.1 that remove recursive cycles while preserving dominance. The cases are:

1. The one and only cycle desired is that from node 0 to itself with symbol 0. This cycle simply zero extends the representations as desired. It is explicitly included by the condition $[V_p \in \mathbb{S}, V_p = 0]$ of line 8.

2. The condition $[V_p \in \mathbb{S}, V_p \neq 0, V_p \notin \mathbb{S}_0]$ of line 12 creates a short cut arc to the zero node if it can be done in one step, i.e., the node's value is in the symbol set. In the example at node -1, condition 2 places the $\overline{1}$ arc but leaves out the 3 arc, thus eliminating the $\overline{1}(03)\cdots(03)_2$ representations which are all dominated by the $(0)\cdots(0)\overline{1}_2$ representation.

3. The condition $[V_p \in \mathbb{S}, V_p \neq 0, V_p \in \mathbb{S}_0]$ of line 10 captures a special case when the node's value is in the $\gamma = 0$ symbol subset, $\mathbb{S}_0 = \{0, 2\}$ [1]. This is shown at node 2 of the example where the 2 shortcut arc is placed to node 0. However, a dominant path through the 0 arc is still possible so the 0 arc is also placed. This ensures the survival of the two correct zero-dominant results 002_2 and $\overline{1}30_2$.

4. The single length cycles at nodes 1 and -2, with arcs $\overline{1}$ and 2 respectively, are excluded by the condition $[s_i \in \mathbb{S}_\gamma, V_c = V_p]$ of line 21.

All other paths must satisfy the condition $[s_i \in \mathbb{S}_\gamma, V_c \neq V_p]$ of line 19 as previously explained in Section 2.7.

[1] See Section 2.7 for a reminder of the subset definitions

4.6 ZERO DOMINANT SET GENERATION EXAMPLE

An example of the ZDS tree generated for the value 23_{10} using the number system $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$, $\beta = 2$ is shown in Figure 4.3. The algorithm proceeds as:

1. The root node (23_{10}) is initially in the parent work list.
2. Since 23 mod 2 = 1, the three symbols from $\mathbb{S}_1 = \{\overline{1}, 1, 3\}$ are used to branch and create the new child nodes for 10_{10}, 11_{10}, and 12_{10}.
3. Their values are all different so no dominance can exist.
4. The parent list is replaced with these new nodes.
5. The next generation is then generated with symbols from $\mathbb{S}_0 = \{0\}$ for nodes 10_{10} and 12_{10} and from $\mathbb{S}_1$ for node 11_{10}.
6. Dominance is then checked and found for two pairs of nodes, with values 5_{10} and 6_{10} respectively.
7. The dominated nodes are removed from the child work list and the others replace the parent list.
8. The algorithm continues until all parent nodes have zero value.

At the termination of the algorithm three nodes remain containing the zero dominant representations $0300\overline{1}_2$, 10031_2, and 10103_2.

4.7 EXTENDING TO HYBRID AND MIXED-RADIX NUMBER SYSTEMS

With little modification the BUILDZDS algorithm can be extended to more exotic number systems. These may include mixed radix systems where β changes as a function of the position. This would simply require β to be updated at each iteration. Similarly, the symbol alphabet may change depending on the position, such as the hybrid signed digit number system [Phatak and Koren, 1994]. A combination of both might also be implemented. The algorithm may also be used as a basis for generating other types of dominant representations with specific properties. An example might be forcing a particular location to have a particular symbol.

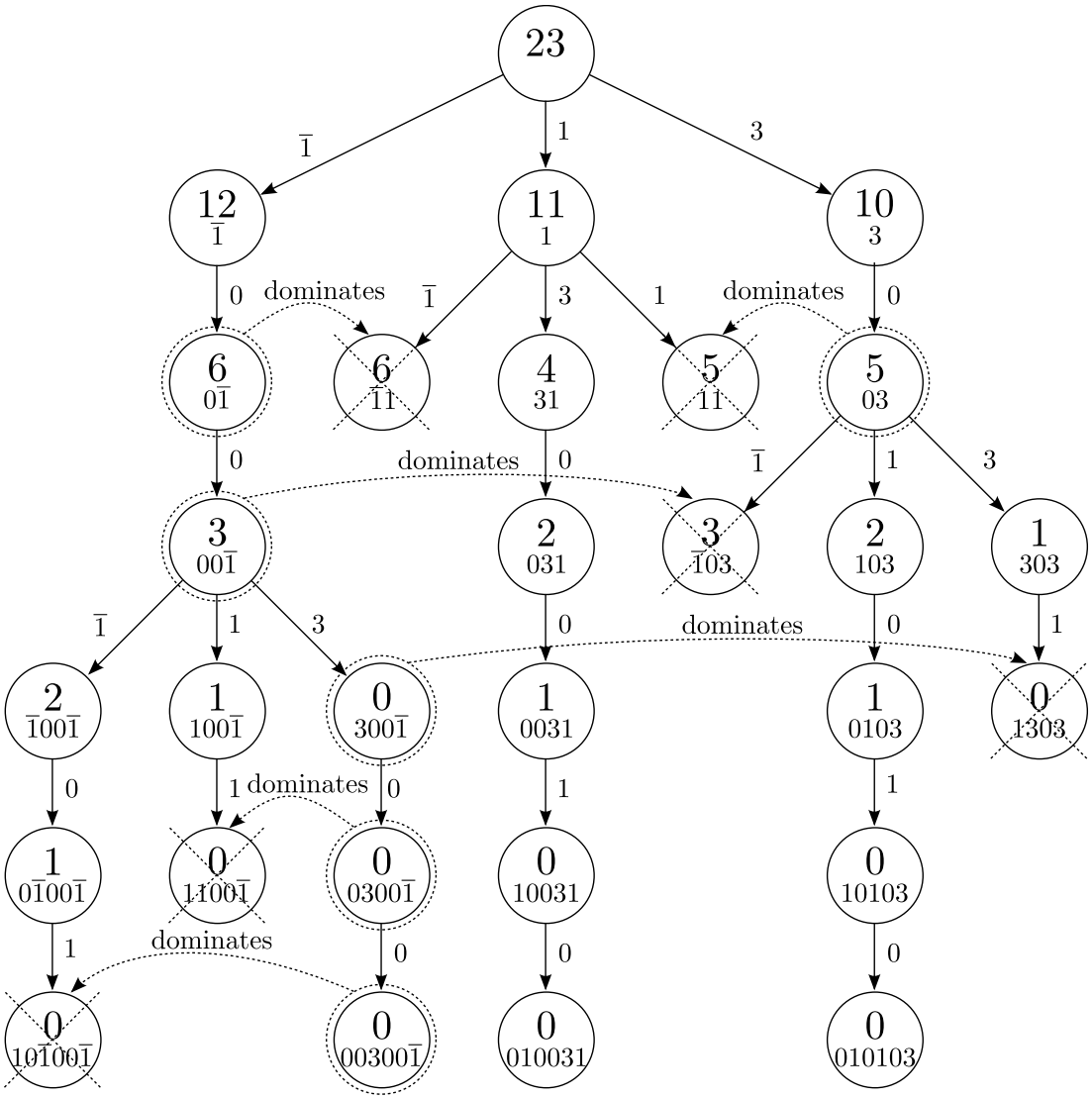


Figure 4.3: Dominant representation tree generated for the value 23_{10}, symbol alphabet $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$, and radix $\beta = 2$. The nodes are labelled with the remaining value to be coded (V_l) and the LSS of the representation to that point. The edges are labelled with symbol (s_i) to be added to the MSS of the representation string. The surviving representations are the zero-dominant set $\{300\overline{1}_2, 10031_2, 10103\}$.

4.8 ZERO DOMINANT SET SUMMARY

The zero-dominance between representations has been defined assuming the zero-symbol is the most desired symbol at each position. For a representation to be zero-dominant, no other possibe representation can have zeros in all the same positions and have zeros in extra positions. Otherwise, the other representation will be dominant and cover the former representation in a Pareto-optimal sense. The surviving representations form the ZDS and each has unique zero placements with all superfluous nonzero symbols removed. The ZDS is a finite cardinality subset of all redundant representations which may have infinite cardinality.

A new algorithm has been presented that searches for the ZDS of any integer value using a branch and bound technique. The number system can have any nonzero integer radix and any symbol alphabet that includes zero. However, with an increasingly large radix the number of zeros present in a value's representation decreases drastically. To maintain an equivalent number of zeros, a significant increase in the symbol alphabet cardinality is required. Therefore, a number system with $\beta = \pm 2$ gives the best results when considering implementation in Boolean logic.

In the best case, when a non-redundant number system is used, the algorithm has a performance that is $O(N)$. In the worst case where the number system is redundant and there is no zero-dominance, the algorithm has a performance of $O(N^2)$ so it is not suitable for on-the-fly recoding of operands. It would be best used to precompute representations sets that in turn can optimise the performance of a calculation kernel, such as a FIR filter. This application is presented in Chapter 8.

Chapter 5

ADDITION WITH REDUNDANT REPRESENTATIONS IN FPGA

A new redundant number system has little practical use for digital signal processing (DSP) unless it can be used in arithmetic. Most arithmetic functions consist of adders as their core computing element. It is the basic building block for many algorithms, including subtraction, multiplication, division, and exponentiation [Mano and Kime, 2004]. Therefore, an adder architecture that accepts redundant number representations is essential.

Knowing the importance of addition, field programmable gate array (FPGA) manufacturers have included special logic and routing in their architectures for binary addition. The adder structure employed is the simplest and smallest, the binary ripple-carry adder [Altera Cyclone III, 2008, Xilinx Spartan-3, 2008]. Ripple-carry adders have the advantage of being compact; implementable in one Altera Cyclone logic element (LE) or half a Xilinx Spartan slice per bit. Their major disadvantage is a long critical path that passes through every cell along the adder. This is called the carry chain. In theory, this makes the delay linearly proportional to the operand width. At moderate to long word lengths this delay path can be the system's performance limitation. A dedicated carry signal path with a low propagation delay is provided within FPGAs to speed up this inherently slow structure, see appendix B.

Traditionally, prefix-tree adders are used in application specific integrated circuit (ASIC) designs to accelerate carry propagation. However, the carry-lookahead logic employed does not map well to the FPGA architecture. No performance improvement is gained over ripple-carry addition unless long word lengths are in use, at which point the logic resource usage has grown impractically large [Vitoroulis and Al-Khalili, 2007]. Some other advanced carry chains for FPGAs have been explored in literature where the carry-logic architecture is redesigned and expanded [Hauck et al., 2000, Frederick and Somani, 2006]; these solutions tend to use a large silicon area, reducing the generic logic density of the device, and as such have not been adopted by the manufacturers.

This chapter begins with a review of binary ripple-carry addition in Sections 5.1 and 5.2. This is followed by an introduction to redundant number system addition in Section 5.3 and how carry-propagation-free addition has previously been achieved in Section 5.4.

Original work on the efficient implementation of fast redundant adders of Xilinx Spartan and Altera Cyclone is presented from Section 5.5. The use of compressor functions for BSD ($\mathbb{S} = \{\overline{1}, 0, 1\}$,$\beta = 2$) and carry-save ($\mathbb{S} = \{0, 1, 2\}$,$\beta = 2$) redundant number addition is previously known. However, the work presented in [Kamp et al., 2009] and repeated in Sections 5.6 through 5.9, shows a novel and resource efficient implementation of 3:2 and greater compressor functions for Spartan 3 and Cyclone III FPGAs. Included is the description of a recoding step that allows the addition of arbitrary symbol alphabets. It is introduced in Section 5.6 and the details of its design method are presented in Section 5.10.

5.1 COMPUTER ARITHMETIC

Computer arithmetic is most commonly performed using the binary number representation. It is easy to understand, easy to implement, and mature. The basic arithmetic entity is the full adder. Arranging full adder entities in regular grids permits the creation of adders and multipliers for binary numbers of any width. Binary arithmetic is slow compared to the speed of the implementing logic. This is because the worst case addition requires the carry chain to propagate a carry signal from the least significant bit (LSB) to the most significant bit (MSB).

There is still a large gap in the performance and area efficiency of arithmetic circuits between ASIC and FPGA. FPGA manufacturers have responded in part by providing fast carry chains and specialised circuitry for implementing ripple-carry style binary adders in their FPGA. This halves the area required by allowing a full adder to be mapped into a single logic block, implementing both the sum and carry signals. Performance is improved by providing fast and dedicated routing for the carry signal between adjacent logic blocks, effectively minimising the carry propagation time. However, the ripple-carry adder is fundamentally slow since the critical delay path is along the carry signals, passing through every logic block. In an ASIC design, advanced carry-chains can be employed such as carry-look-ahead, carry-select, or parallel prefix-tree designs like the Brent-Kung adder to improve the delay of parallel adders [Nagendra et al., 1996], see Appendix A for a description of these methods.

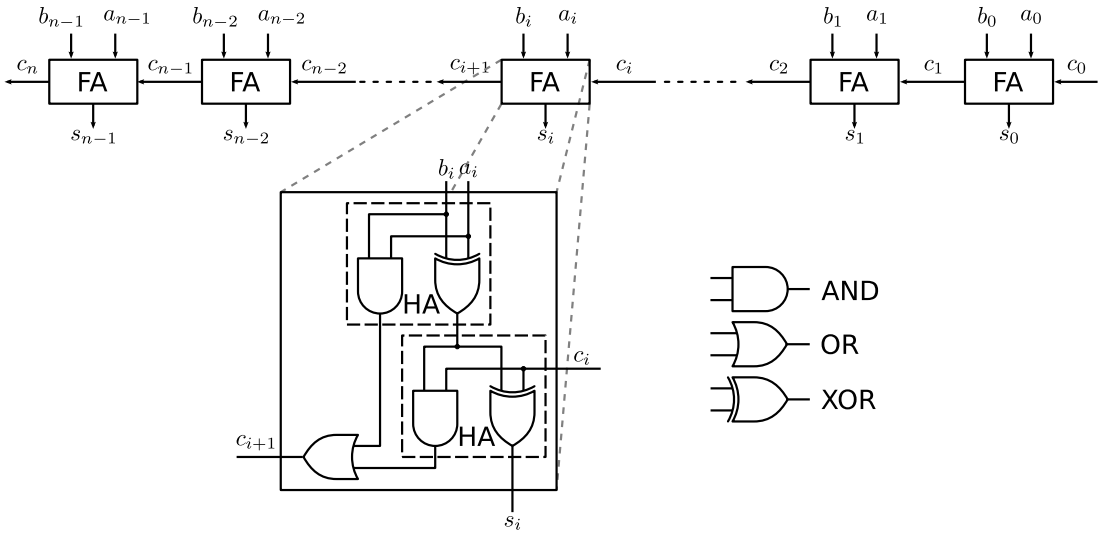


Figure 5.1: Binary ripple-carry adder architecture showing carry chain between full adders (FA) and the gate detail of full adders and half adders (HA).

5.2 BINARY ADDITION

Addition takes two numbers and sums them to give a single result with a width one greater than the maximum operand width. This increased width is because the maximum resulting value can be twice as large as the input values. Addition of binary numbers can be achieved with a cascaded structure of full adders (FA) constructed from pairs of half adders (HA) as shown in Figure 5.1.

An example of a binary addition that shows all inputs cases, the algebraic sum of 108_{10} and 90_{10}, is shown in Figure 5.2. The bits are evaluated from right to left and the logic generates a sum and a carry for each symbol place. The carry from the previous full adder is input to the next full adder in the chain. This means that the next input bit pair cannot be evaluated until the previous bit pair has been resolved into a sum and carry. The transport of the carry through each of the full adders is referred to as carry chain propagation. The binary addition result, S, is not valid until the carry chain propagation, C, has terminated at the most significant bit (MSB) as shown in Figure 5.3. The total calculation time is split into time steps (t_x) that are equal to the propagation delay of an individual full adder circuit.

The rippling effect on the carry as it propagates through the full adders gives this basic adder structure the name ripple-carry. The example shows that the worst case time to complete a binary addition in a ripple-carry adder is linearly proportional to the input width N. In big-oh notation, ripple-carry binary addition calculation speed is $O(N)$. The required logic size is also linearly proportional to the number of bits, $O(N)$,

$$
\begin{array}{rcccccccc}
C = & \overset{1}{\curvearrowleft} & \overset{1}{\curvearrowleft} & \overset{1}{\curvearrowleft} & \overset{1}{\curvearrowleft} & & & & \\
X = & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \quad = 108_{10} \\
+Y = & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \quad = 90_{10} \\
\hline
S = & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \quad = 198_{10}
\end{array}
$$

Figure 5.2: The binary addition of 108_{10} and 90_{10}, showing all eight input variations for bits of X and Y and the carry C.

$$
\begin{array}{rcccccc}
X = & 0 & 1 & 1 & 1 & 1 & = 15_{10} \\
+Y = & 0 & 0 & 0 & 0 & 1 & = 1_{10} \\
\hline
C_{t_1} = & & & & \overset{1}{\curvearrowleft} & & \\
S_{t_1} = & 0 & 1 & 1 & 1 & 0 & = 14_{10} \\
\hline
C_{t_2} = & & & \overset{1}{\curvearrowleft} & & & \\
S_{t_2} = & 0 & 1 & 1 & 0 & 0 & = 12_{10} \\
\hline
C_{t_3} = & & \overset{1}{\curvearrowleft} & & & & \\
S_{t_3} = & 0 & 1 & 0 & 0 & 0 & = 8_{10} \\
\hline
C_{t_4} = & \overset{1}{\curvearrowleft} & & & & & \\
S_{t_4} = & 0 & 0 & 0 & 0 & 0 & = 0_{10} \\
\hline
C_{t_5} = & & & & & & \\
S_{t_5} = & 1 & 0 & 0 & 0 & 0 & = 16_{10} \\
\hline\hline
X + Y = & 1 & 0 & 0 & 0 & 0 & = 16_{10}
\end{array}
$$

Figure 5.3: The binary addition of 15_{10} and 1_{10} evaluated from LSB to MSB. The calculation is time sliced into 5 parts where the carry propagates. This illustrates the worst case situation where the carry signal must propagate the entire length of the word.

as one full adder is required for every pair of input bits. The ripple-carry adder will be considered the baseline benchmark for comparing other adder structures in terms of propagation delay and logic size.

5.3 REDUNDANT NUMBER ADDITION

The use of redundant number systems can significantly improve computational performance in numerically intensive applications (see Chapter 3). However, their implementation is fundamentally expensive because multiple signals are needed to encode each symbol (see Chapter 2). Normally an addition of two operands is evaluated from least significant symbol (LSS) to most significant symbol (MSS). An intermediate sum is calculated at each symbol position by adding the input operand symbols and any carry-in symbols, $U_{\langle i \rangle} = A_{\langle i \rangle} + B_{\langle i \rangle} + C_{\langle i \rangle}$. The intermediate sum is translated into a sum symbol in the same position and carry symbols in higher positions.

All addition requires the use of carry symbols, sometimes called transfer symbols. Without these the adder is just a grouping of symbols with the same weight and possibly a recoding. If used in an accumulation, the output symbol alphabet would grow with each subsequent addition. Carry symbols alleviate this growth by transporting value further up the positional system where, with each step, the carry symbol's value is divided by β. A carry symbol may immediately skip k positions further up the word and consequently its value is divided by β^k.

Using the ripple-carry addition structure common for binary adders, a BSD adder can be constructed as in Figure 5.4. The operand symbols $A_{\langle i \rangle}$ and $B_{\langle i \rangle}$ have the symbol alphabet $\{\overline{1}, 0, 1\}$. When adding these two symbols it gives the half-intermediate sum set of $\{\overline{2}, \overline{1}, 0, 1, 2\}$. Since $\overline{2}$ and 2 are not in the BSD output alphabet, the carry-out symbol must have the alphabet $\{\overline{2}, 0, 2\}$ to transfer their values to the next position. This reduces the carry symbol's values by a factor of $\beta = 2$ to give the carry-in alphabet $\{\overline{1}, 0, 1\}$. This carry-in alphabet is added to the half-intermediate sum set. The resulting $3^3 = 27$ combinations of two operands symbols and a carry-in symbol gives the full intermediate sum alphabet of $\{\overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3\}$. Subtracting the carry-out symbol alphabet $\{\overline{2}, 0, 2\}$ leaves a sum-out alphabet of $\{\overline{1}, 0, 1\}$, the same as the input operands.

5.4 CARRY-PROPAGATION-FREE ADDITION

Carry-propagation-free addition is where the carry-in symbol to a position does not affect a change in the carry-out symbol. Alternatively, the carry-out symbol is generated

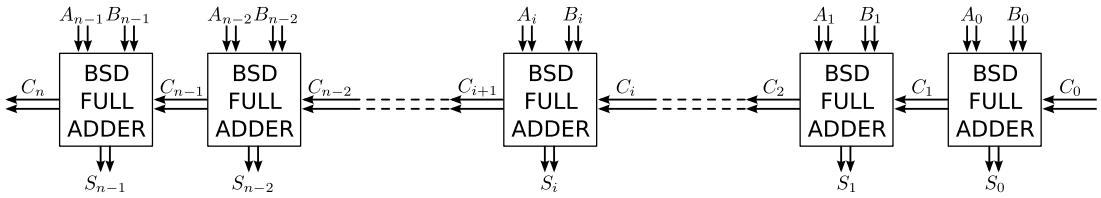


Figure 5.4: Binary signed digit (BSD) ripple-carry adder structure.

independently of the carry-in symbol. This means that a change in one carry symbol cannot propagate up the word, giving the addition a constant delay irrespective of the word length. This constant-time addition can be achieved when the output number system is redundant. Several schemes have been proposed to perform carry-propagation-free addition, often simply called carry-free[1] addition. Other descriptive names may be 'totally parallel' or 'bit parallel' addition, which try to emphasise the simultaneous evaluation of all bits in the word, as opposed to the operands being presented to the adder as parallel signals but using serial evaluation from LSS to MSS [Parhami, 1988].

Signed digit arithmetic was first introduced by Avizienis [1961]. In his initial paper, addition was described as carry-free only for radii greater than two, thus excluding BSD. For BSD, to avoid carrying a symbol into the next position the output symbol alphabet must include the symbols $\overline{2}$ and 2, for the symbol additions of $\overline{1}+\overline{1}$ or $1+1$ respectively. Thus it was thought that it is not possible to resolve a BSD addition to the original symbol set $\{\overline{1}, 0, 1\}$ without a carry chain. However, later work has addressed this problem by eliminating carry propagation in one of several ways:

1. Select a particular redundant representations for each operand that avoids the carry propagation conditions, i.e., make sure that a zero symbol follows every nonzero symbol in both operands [Parhami, 1988]. If a carry is generated, possible only when two nonzero symbols are added, then it will fall into the next position which is guaranteed to be zero. This effectively captures the carry and avoids propagation. For a BSD addition the redundant representations chosen are the canonical signed digit (CSD) representations discussed in Section 2.4.1.

2. Peek at the operand symbols in lower positions, the 'right context', to work out the possible carry-in signals. This extra information allows the choice of the carry-out symbol such that the carry-in symbol only affects the sum symbol [Kuninobu et al., 1987a, Phatak et al., 2001].

3. Use multiple levels within the adder, summing only two symbols at a time to give sum and carry symbols to be added at the next level [González and Mazumder,

[1] "Carry-free" is misleading; carry symbols still exist and are necessary in these adder designs, it is the propagation of carries that is avoided.

$$
\begin{array}{rccccc|l}
A = & 1 & 0 & 1 & 0 & \overline{1} & = 19_{10} \\
+B = & 0 & 0 & 1 & 0 & \overline{1} & = 3_{10} \\
\hline
C_{t_1} = & & & \overset{1}{\curvearrowleft} & & \overset{\overline{1}}{\curvearrowleft} & \\
S_{t_1} = & 1 & 0 & 0 & 0 & 0 & = 16_{10} \\
\hline
C_{t_2} = & & & & & & \\
S_{t_2} = & 1 & 1 & 0 & \overline{1} & 0 & = 22_{10} \\
\hline\hline
A + B = & 1 & 1 & 0 & \overline{1} & 0 & = 22_{10}
\end{array}
$$

Figure 5.5: An example of canonical signed digit (CSD) ripple-carry addition. A carry symbol is only generated when two identical nonzero symbols are added. The CSD encoding ensures the position the carry falls into is occupied by zeros, thus carry propagation is eliminated. Note the result is in BSD.

2000]. This completely removes any possible carry propagation path.

Further details on the first two methods are given in Sections 5.4.1 and 5.4.2 respectively. The third method works most generally and maps best to the FPGA architecture. It is used in the rest of the chapter for a novel FPGA implementation.

5.4.1 CSD addition

Performing a ripple-carry addition in BSD provides no performance gain over binary and is less resource efficient in digital hardware. Recoding the operands to CSD allows carry-propagation-free addition to be performed using the ripple-carry hardware structure of Figure 5.4. By requiring the operands to be in CSD form, where any nonzero-symbol is followed by a zero-symbol, any carry generated is guaranteed to fall into a position occupied by a zero-symbol. This effectively removes the condition that causes carry propagation, leaving only the condition for carry generation. If a carry is generated in position i then it is terminated in position $i + 1$. This capture of the carry is shown in the example of Figure 5.5.

The output is guaranteed to be stable after the carries have propagated one step. However, the output of the addition will be in BSD rather than canonical form. For the BSD result to be used in a subsequent calculation and maintain the same speed benefit it must be recoded back into CSD form. The conversion of a binary or BSD representation into CSD requires a signal to propagate the length of the word. This is slow and is the primary draw back of this adder method.

5.4.2 BSD look-back addition

The second method of redundant addition is to *look-back* at the operand symbols in lower positions, the *right context*. This extra information is used to predict the range of the carry-in, then choose the sum and carry-out such that a nonzero carry-in need only modify the sum symbol [Kuninobu et al., 1987a, Phatak et al., 2001]. For BSD, only the symbols in the postion immediately below need to be inspected.

Look-back addition of two BSD numbers to avoid carry-propagation is a two stage process:

1. Determine the carry-out symbol $C_{\langle i+1\rangle}$ and the intermediate sum symbol $U_{\langle i\rangle}$ from the sum of the symbols $(A_{\langle i\rangle} + B_{\langle i\rangle})$ and the right context $(A_{\langle i-1\rangle}, B_{\langle i-1\rangle})$.

   (a) If either of the right context symbols $A_{\langle i-1\rangle}, B_{\langle i-1\rangle}$ is $\overline{1}$ then the carry-in $C_{\langle i\rangle}$ will be in the negative range $\{\overline{1}, 0\}$. $U_{\langle i\rangle}$ is restricted to the positive range $\{0, 1\}$.

   (b) Similarly if the right context causes a positive ranged carry-in, $U_{\langle i\rangle}$ is restricted to be in the negative range.

   The values of $U_{\langle i\rangle}$ and $C_{\langle i+1\rangle}$ are chosen so that $C_{\langle i+1\rangle} + U_{\langle i\rangle} = A_{\langle i\rangle} + B_{\langle i\rangle}$ and $U_{\langle i\rangle}$ is in the correct range. The truth table for this stage is given in Table 5.1.

2. Determine the output sum symbol $S_{\langle i\rangle}$ as the sum of $C_{\langle i\rangle}$ and $U_{\langle i\rangle}$ as shown in Table 5.1.

The range restriction of the first stage ensures that the sum symbol will be in the BSD symbol alphabet $\{\overline{1}, 0, 1\}$. There is never any need to change $C_{\langle i+1\rangle}$ and thus there is no carry propagation.

5.5 EFFICIENT IMPLEMENTATION OF FAST REDUNDANT NUMBER ADDERS FOR LONG WORD-LENGTHS IN FPGA

The following sections present efficient redundant number adder circuits specifically targeted to the low-cost FPGA architectures of the Xilinx Spartan 3 [Xilinx Spartan-3, 2008] and the Altera Cyclone III [Altera Cyclone III, 2008]. The carry logic and fast carry chains specially provided for binary ripple-carry addition are repurposed to serve the new adders. Section 5.6 describes the general structure of a redundant number adder that maps well into FPGA architecture. Section 5.8 presents the logical implementation of compressor functions, the building blocks of redundant number adders, and show

Table 5.1: Truth table for a look-back binary signed digit (BSD) adder. × means "don't care".

Symbol Sum	Right Context		Carry-out	Inter-Sum	$S_{\langle i \rangle} = C_{\langle i \rangle} + U_{\langle i \rangle}$ given:		
$A_{\langle i \rangle} + B_{\langle i \rangle}$	$A_{\langle i-1 \rangle}$	$B_{\langle i-1 \rangle}$	$C_{\langle i+1 \rangle}$	$U_{\langle i \rangle}$	$C_{\langle i \rangle} = \overline{1}$	$C_{\langle i \rangle} = 0$	$C_{\langle i \rangle} = 1$
-2	×	×	$\overline{1}$	0	$\overline{1}$	0	1
-1	×	$\overline{1}$	$\overline{1}$	1	0	1	–
-1	$\overline{1}$	×	$\overline{1}$	1	0	1	–
-1	otherwise		0	$\overline{1}$	–	$\overline{1}$	0
0	×	×	0	0	$\overline{1}$	0	1
1	×	$\overline{1}$	0	1	0	1	–
1	$\overline{1}$	×	0	1	0	1	–
1	otherwise		1	$\overline{1}$	–	$\overline{1}$	0
2	×	×	1	0	$\overline{1}$	0	1

how they can be implemented on the Spartan 3. The implementation in a Cyclone III is then presented in Section 5.9. These circuits use the redundancy in the result's representation to eliminate carry propagation, providing near constant addition delay irrespective of the operand width. This is confirmed experimentally and shown to outperform the architecture optimised binary ripple-carry adders at long word lengths. The performance and resource usage results for the Spartan 3 are given in Section 5.8.4 and for the Cyclone III in 5.9.2. The critical path delay crossovers for the binary and BSD adders are at widths of 44 and 24 symbols, using only 2 and 3 times the number of look-up tables on the Spartan 3 and Cyclone III respectively. Theoretically fast binary prefix-tree adders do not compare favourably at any width, because they either have a larger delay or they consume more logic resources.

5.6 REDUNDANT ADDITION USING COMPRESSOR FUNCTIONS

A compressor function of rank r takes a number of input symbols of the same rank (ordinal position in the representation) and sums their values. The output is a new set of symbols, with a variety of ranks greater or equal to r and fewer in number than the inputs. A common example is the binary full adder circuit that takes three positive binary bits of rank r (A, B, and carry-in) and compresses these to two positive binary output bits, a "sum" bit of rank r and a "carry" bit of rank $r + 1$. It is considered a 3:2 compressor.

A general structure for redundant number addition takes the form of layered

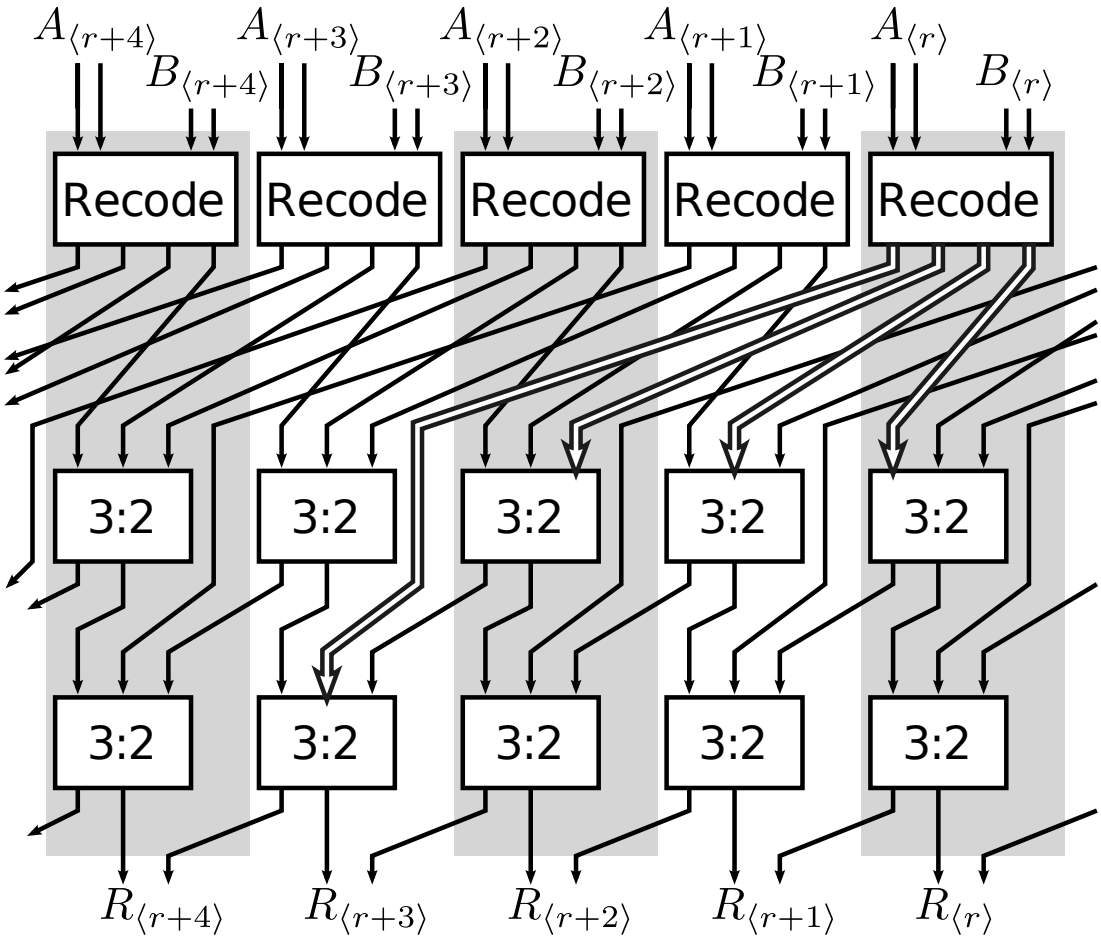


Figure 5.6: Generalised redundant adder structure. The signals from the recoding of $A_{\langle r\rangle} + B_{\langle r\rangle}$ are shown with bold arrows to help differentiate them from the repeating background.

compressor functions. This is usually in the form of Figure 5.6. Note that for a redundant number system each symbol needs to be encoded onto two or more digital signals. In the example of Figure 5.6, the two operands use symbols encoded on two digital signals each. A redundant number system with a radix of two and a symbol alphabet with maximum cardinality of four provides a good match to low-cost FPGAs with 4-input lookup tables (4-LUTs). This allows two symbols, encoded on two bits each, to be operated on without having to build a multiple LUT cone of logic for each output bit (cf. sections C.1 and C.2).

The first level of logic in Figure 5.6 generates an intermediate sum symbol from the sum of two input symbols. The intermediate sum symbol is then recoded into a simpler format. For radix-2, the output format is a mini-representation that has a symbol in several ranks from either of the alphabets $\mathbb{P} = \{0, 1\}$ or $\mathbb{M} = \{\overline{1}, 0\}$.[2]

[2]Read $\mathbb{P}$ as plus alphabet and $\mathbb{M}$ as minus alphabet.

The intermediate sum mini-representation at each position is non-redundant since the cardinality of the output symbol alphabets, $|\mathbb{P}|$ and $|\mathbb{M}|$, is equal to the radix of two. The mini-representations consist of a sum symbol, $S_{\langle r \rangle}$, of rank r and several transfer or carry symbols, $C_{\langle r+k \rangle}$, of higher rank that carry a value to a position k places higher in the addition. An example recoding for the sum of symbols from the alphabets $\{\overline{1}, 0, 1\}$ and $\{\overline{1}, 0, 1, 3\}$ is shown in Table 5.2.

For certain symbol alphabets that are already appropriately encoded onto their signals, this first recoding level can be achieved solely with routing. Some examples of free encodings are shown in Table 5.3. This may result in duplicate encodings for a symbol, such as in BSD where both $0_{\langle r \rangle}0_{\langle r \rangle}$ and $1_{\langle r \rangle}\overline{1}_{\langle r \rangle}$ are valid encodings for the zero symbol. Two examples of symbol alphabets that cannot be appropriately encoded are $\{\overline{1}, 0, 1, 3\}$ and $\{0, 1, 3, 5\}$, as both only require two signals but no two integers can be summed with themselves or zero to give all values in the symbol alphabet.

At subsequent levels of the addition, all the transfer and sum symbols of the same rank are grouped into threes. Each group is reduced to a new transfer and sum symbol using a 3:2 compressor. The 3:2 compressor sums three inputs of $\mathbb{P}$ or $\mathbb{M}$ of rank r to give two outputs, a transfer symbol of rank $r + 1$, and a sum symbol of rank r, appropriately from either $\mathbb{P}$ or $\mathbb{M}$.

The compressors can be written as a list of input and output symbol alphabets separated with a colon. The alphabets are augmented with their rank in subscript. The four permutations of three inputs with their two appropriate outputs are:

1. $\mathbb{P}_{\langle r \rangle}\mathbb{P}_{\langle r \rangle}\mathbb{P}_{\langle r \rangle}{:}\mathbb{P}_{\langle r+1 \rangle}\mathbb{P}_{\langle r \rangle}$,
2. $\mathbb{P}_{\langle r \rangle}\mathbb{P}_{\langle r \rangle}\mathbb{M}_{\langle r \rangle}{:}\mathbb{P}_{\langle r+1 \rangle}\mathbb{M}_{\langle r \rangle}$,
3. $\mathbb{M}_{\langle r \rangle}\mathbb{M}_{\langle r \rangle}\mathbb{P}_{\langle r \rangle}{:}\mathbb{M}_{\langle r+1 \rangle}\mathbb{P}_{\langle r \rangle}$, and
4. $\mathbb{M}_{\langle r \rangle}\mathbb{M}_{\langle r \rangle}\mathbb{M}_{\langle r \rangle}{:}\mathbb{M}_{\langle r+1 \rangle}\mathbb{M}_{\langle r \rangle}$.

Their logic is shown in Table 5.4 and Table 5.5. The regular subscripted rank notation $\langle r \rangle$ is dropped to conserve space.

Any left over symbols that do not make a complete group of three are left to be combined at the next level. As many levels of 3:2 compressors are employed as is necessary to reduce the addition to two symbols per position. The remaining two symbols can be interpreted as a single signed digit, carry-save, or borrow-save symbol. This is similar to a Dadda addition tree used in ASICs for partial product accumulation in parallel multiplication [Dadda, 1977].

Table 5.2: Level 1 compressor to generate the radix-2 intermediate sum of symbol alphabets $\{\overline{1}, 0, 1\}$ and $\{\overline{1}, 0, 1, 3\}$.

Symbols	Sum	Output ($C_{\langle r+2\rangle}C_{\langle r+1\rangle}S_{\langle r\rangle}$)
$\overline{1}+\overline{1}$	-2	$0\overline{1}0_2$
$\overline{1}+0$	-1	$0\overline{1}1_2$
$\overline{1}+1$	0	000_2
$\overline{1}+3$	2	$1\overline{1}0_2$
$0+\overline{1}$	-1	$0\overline{1}1_2$
$0+0$	0	000_2
$0+1$	1	001_2
$0+3$	3	$1\overline{1}1_2$
$1+\overline{1}$	0	000_2
$1+0$	1	001_2
$1+1$	2	$1\overline{1}0_2$
$1+3$	4	100_2

Table 5.3: Examples of redundant number systems that can be encoded such that the signals can be separated into $\mathbb{P} = \{0, 1\}$ or $\mathbb{M} = \{\overline{1}, 0\}$ symbols at different ranks.

Redundant Number System	Alphabet	Encoding
Binary Signed Digit	$\{\overline{1}, 0, 1\}$	$< \mathbb{P}_{\langle r\rangle}\mathbb{M}_{\langle r\rangle} >$
Carry-Save	$\{0, 1, 2\}$	$< \mathbb{P}_{\langle r\rangle}\mathbb{P}_{\langle r\rangle} >$
Borrow-Save	$\{\overline{2}, \overline{1}, 0\}$	$< \mathbb{M}_{\langle r\rangle}\mathbb{M}_{\langle r\rangle} >$
Carry Double-Save	$\{0, 1, 2, 3\}$	$< \mathbb{P}_{\langle r+1\rangle}\mathbb{P}_{\langle r\rangle} >$
Borrow Double-Save	$\{\overline{3}, \overline{2}, \overline{1}, 0\}$	$< \mathbb{M}_{\langle r+1\rangle}\mathbb{M}_{\langle r\rangle} >$
Asymmetric Signed Digit	$\{\overline{1}, 0, 1, 2\}$	$< \mathbb{P}_{\langle r+1\rangle}\mathbb{M}_{\langle r\rangle} >$
Asymmetric Signed Digit	$\{\overline{2}, \overline{1}, 0, 1\}$	$< \mathbb{M}_{\langle r+1\rangle}\mathbb{P}_{\langle r\rangle} >$

Table 5.4: Truth Table for the 3:2 compressors of type $\mathbb{PPP}$:$\mathbb{PP}$. This is a binary full adder. The truth table for $\mathbb{MMM}$:$\mathbb{MM}$ is similar with switched signs.

Input Symbols $\mathbb{PPP}$ ($\mathbb{MMM}$)	Intermediate Sum	Output $\mathbb{PP}$ ($\mathbb{MM}$)
$0+0+0$	0	00_2
$0+0+1$	1	01_2
$0+1+0$	1	01_2
$0+1+1$	2	10_2
$1+0+0$	1	01_2
$1+0+1$	2	10_2
$1+1+0$	2	10_2
$1+1+1$	3	11_2

Table 5.5: Truth Table for the 3:2 compressors of type $\mathbb{PPM}$:$\mathbb{PM}$. The truth table for $\mathbb{MMP}$:$\mathbb{MP}$ is similar with switched signs.

Input Symbols $\mathbb{PPM}$ ($\mathbb{MMP}$)	Intermediate Sum	Output $\mathbb{PM}$ ($\mathbb{MP}$)
$0+0+0$	0	00_2
$0+0+\overline{1}$	-1	$0\overline{1}_2$
$0+1+0$	1	$1\overline{1}_2$
$0+1+\overline{1}$	0	00_2
$1+0+0$	1	$1\overline{1}_2$
$1+0+\overline{1}$	0	00_2
$1+1+0$	2	10_2
$1+1+\overline{1}$	1	$1\overline{1}_2$

Finally, if a non-redundant binary result is required, a carry signal must be allowed to propagate the full word length. This is covered further in Section 6.2. When using a ripple-carry structure this usually takes a time linearly proportional to the word length. This may seem to negate the purpose of using a constant time adder. However, a calculation that contains many addends or iterations such as accumulation, can use the resulting redundant number system internally and convert to binary only once at the end. For example, a 32-bit multiplier requires up to 31 additions of partial products. The time overhead of the final conversion to binary is amortised over the 31 fast constant time additions. Furthermore, in a finite impulse response (FIR) filter, keeping the multiplication results as a redundant number amortises over many more additions. A FIR filter with 21 taps requires up to $32 \times 21 - 1 = 671$ partial product additions.

5.7 FPGA IMPLEMENTATION

Mapping compressor structures to FPGAs has been studied previously in the context of building multi-operand binary addition trees. The solution proposed by Niaki et al. [2008] is to include additional logic cells that efficiently implement configurable compressor trees or counter arrays as their primary function. This reduces the silicon area available for general LUT based logic blocks. Parandeh-Afshar et al. [2008] proposed the duplication of the carry chain in an Altera Stratix II to allow it to be configured as a 6:2 compressor. These compressor trees or counter arrays can efficiently sum many binary operands. However, they are unsuitable for redundant number system addition because the input and output signals are assumed to be binary, not an arbitrary symbol alphabet.

The first level of the redundant adder performs a recoding from the two input symbols of two bits to k outputs of type $\mathbb{P}$ or $\mathbb{M}$. Since in total four signals are used for the input symbols and k outputs generated, the logic potential of k 4-LUTs is maximally utilised for the recoding. The next levels are constructed with 3:2 compressors of four flavours. These can be paired into two logically identical compressors. These need to be treated separately for implementation in the Xilinx Spartan-3 but the Altera Cyclone III architecture allows them to be treated similarly as discussed in Section 5.9.

5.8 SPARTAN 3 IMPLEMENTATION

This section considers how the compressor circuits of Section 5.6 can be mapped on to the Xilinx Spartan 3 FPGA architecture. A behavioural circuit description results

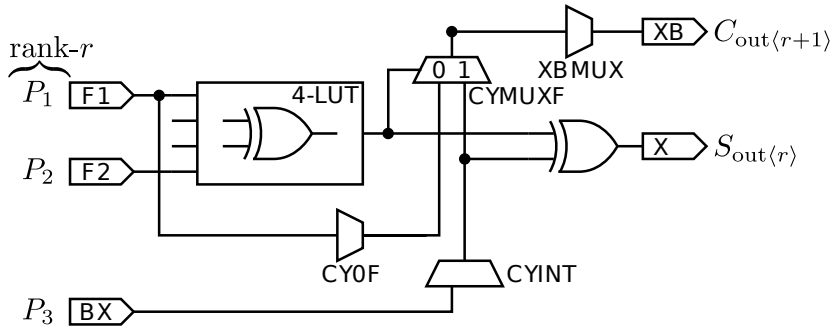


Figure 5.7: Implementation details of a ℙℙℙ:ℙℙ compressor in the lower slice of a Xilinx Spartan 3 FPGA.

in a two 4-LUT synthesis result; one 4-LUT for each of the carry and sum outputs. Instantiating the primitive components in a slice using a structural circuit description halves the LUT usage. One 3:2 compressor of either flavour can be implemented in half a slice.

5.8.1 ℙℙℙ:ℙℙ compressor in Spartan 3

The ℙℙℙ:ℙℙ (𝕄𝕄𝕄:𝕄𝕄) compressor is simply a binary full adder as shown in Figure 5.7. As such, this should map efficiently onto the special logic provided for binary addition using (5.1) and (5.2). One signal is mapped to the carry-in and the other two as binary operand bits. The lower slice can initiate the carry chain with one of the input signals thought the BX wire. The carry-out signal can be routed out through the XB wire. The boolean equations are:

$$C_{\text{out}} = (P_1 \oplus P_2)P_3 + \overline{(P_1 \oplus P_2)}P_1, \tag{5.1}$$

$$S_{\text{out}} = (P_1 \oplus P_2) \oplus P_3. \tag{5.2}$$

5.8.2 ℙℙ𝕄:ℙ𝕄 compressor in Spartan 3

The ℙℙ𝕄:ℙ𝕄 (𝕄𝕄ℙ:𝕄ℙ) compressor has slightly different logic for the carry signal as shown in Table 5.5. The Boolean equations are modified to,

$$C_{\text{out}} = \overline{(P_1 \oplus M_2)}P_3 + (P_1 \oplus M_2)P_1, \tag{5.3}$$

$$\overline{S_{\text{out}}} = \overline{(P_1 \oplus M_2)} \oplus P_3. \tag{5.4}$$

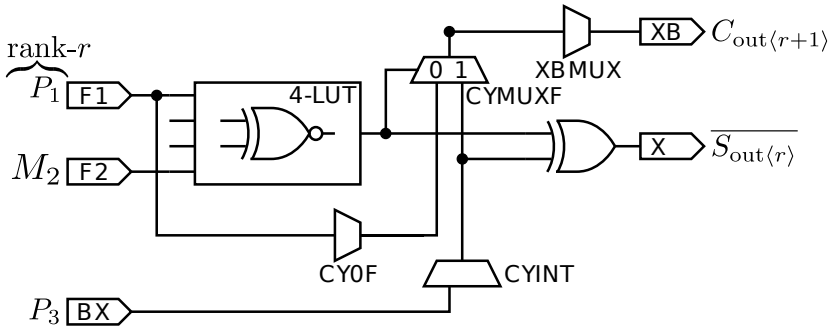


Figure 5.8: Implementation details of a $\mathbb{PPM}$:$\mathbb{PM}$ compressor in the lower slice of a Xilinx Spartan 3 FPGA.

Note that the S_{out} signal is inverted. These equations are reflected in the circuit implementation of Figure 5.8.

5.8.3 Two level 4:2 compressors in Spartan 3

The upper half of a slice cannot initiate the carry chain from the external interconnect, it must initiate from the carry-out of the lower slice. This limits the use of the upper slice as a general 3:2 compressor and relegates it to a lower level compressor in the adder, where the carry-in signal can be a carry output from a previous level compressor.

Connecting two 3:2 compressors of the same rank in series gives a 4:2 compressor as shown by the dotted boundary in Figure 5.9. The name of a 4:2 compressor (and larger compressors) is misleading as it actually has five inputs and three outputs. The extra input is a carry-in and the extra output is a carry-out at the intermediate stage. It is worth noting that no carry propagation can occur since no signal path exists between the extra carry-in and the carry-out signals.

To utilise the upper half of the slice as a 3:2 compressor, the carry-in signal must be used as an input. Pairing a 3:2 compressor of rank r in the first level with a 3:2 compressor of rank $r + 1$ in the second level allows the upper slice to be used as shown by the shaded background in Figure 5.9. This means that two 3:2 compressors can be implemented per slice, as shown in Figure 5.10.

5.8.4 BSD adder performance in Spartan 3

A series of experiments were run to measure the BSD adder performance. The 4:2 compressor based BSD adder was written in structural VHDL[3] using Xilinx primitive

[3]VHDL – VHSIC (very high speed intergrated circuit) hardware description language.

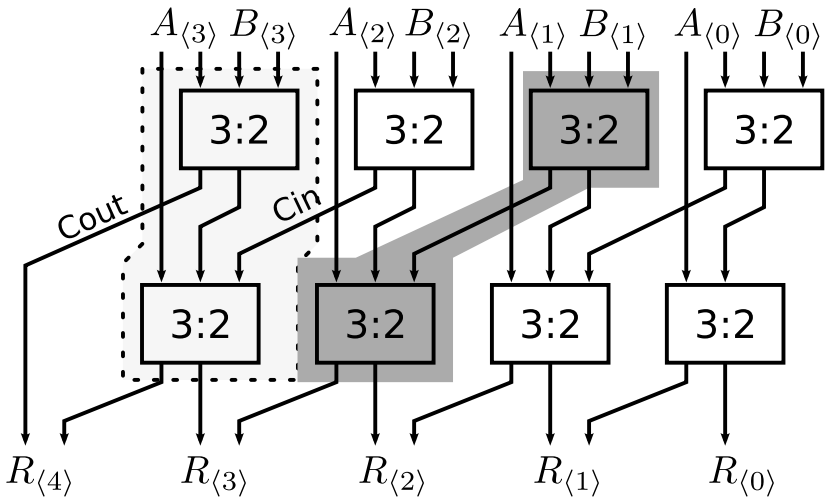


Figure 5.9: A two level redundant number system adder with operands A and B. Each operand symbol is encoded on a pair of digital signals. The two digital outputs are paired to give result symbols $R_{\langle i\rangle}$. The dotted line outlines a 4:2 compressor and the shaded background represents the parts of the 4:2 compressor mapped onto a single Spartan 3 slice.

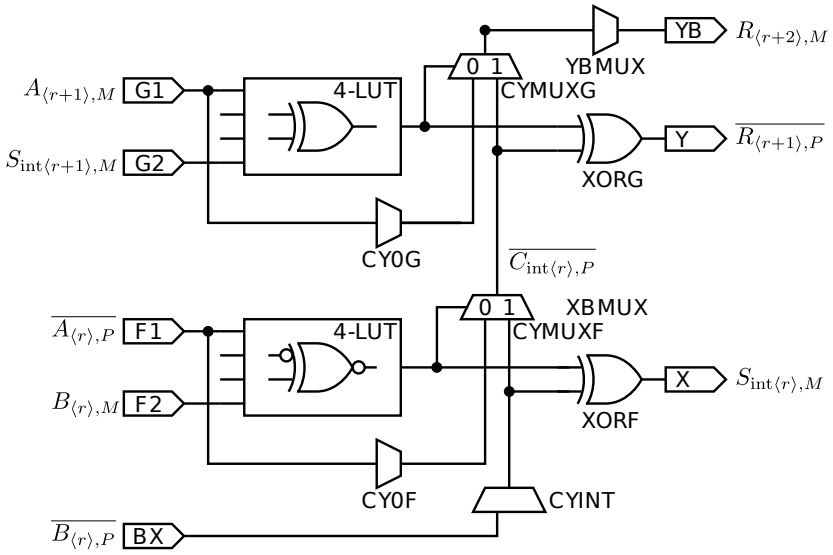


Figure 5.10: A 4:2 compressor function made from a $\mathbb{PPM}$:$\mathbb{PM}$ compressor and a $\mathbb{MMP}$:$\mathbb{MP}$ compressor mapped to a single Xilinx Slice to implement two half columns of a binary signed digit adder.

components. Behavioural models would not synthesise to correctly use the carry logic, resulting in a larger and slower implementation. For each experiment the adder length, N, was varied as powers of two from 8 to 2048 symbols. The adder was placed between input and output registers clocked by the same source. The adders were then synthesised, placed, and routed using Xilinx ISE 10.1 on a Spartan 3 XC3S1500 device. The tool's static timing analysis was used as the measure of performance. Each experiment consisted of 100 trials, one for each 'cost table'[4] in the mapping software, to measure the expected spread of performance. The performance of the BSD adder is shown in Figure 5.11. The slight decline in performance is attributed to the increased routing delay as the number of signals grows with the operand width, and increased clock skew between flip-flops as the registers also grow with the operand width.

For comparison, the same experiment was run for a simple inferred binary ripple-carry adder and a binary Brent-Kung prefix-tree adder [Brent and Kung, 1982]. The Xilinx synthesis tools automatically split the carry chain when the word length exceeds the height of a slice column. The resulting performance curves are plotted alongside the BSD adder in Figure 5.11. The 4:2 compressor design and the ripple-carry adders have similar performance at widths of approximately $N = 44$. The Brent-Kung adder only begins to outperform the ripple-carry adder at widths greater than 128.

A width N binary ripple-carry adder uses N LUTs spread over $\lceil N/2 \rceil$ slices. The area used by the BSD adder is two LUTs per symbol position, twice that of a binary-ripple-carry adder. A width N BSD adder uses $2N$ LUTs spread over $N + 1$ slices. The Brent-Kung adder is comparatively expensive at greater than $5N$ LUTs. The LUT usage for each of the adder designs are given in Table 5.6. It should be noted that the Brent-Kung adder is the least resource intensive of the fast parallel prefix-tree binary adder designs tested by Vitoroulis and Al-Khalili [2007].

A pipelined binary ripple-carry adder with P stages could be used to break the carry propagation chain of long word lengths and improve throughput. However, the latency of the result is still $O(N)$ and PN half slices are required.

Figure 5.11 does not compare completely equivalent adders; it ignores the conversion required from binary to BSD at the input, and from BSD to binary at the output. If binary is a subset of the redundant number system being utilised at the input then the conversion time is obviously constant, requiring only a remapping of the binary bits to the digital symbol encoding used for the redundant number system symbol alphabet. Often this can be done for free if the digital encoding for the symbols is chosen such that the symbols 0 and 1 differ in only one bit. The reverse conversion, from the redundant

[4]The chosen cost table (an integer between 1 and 100) adjusts the balance between parameters used by the mapping and fitting algorithms of the Xilinx tool chain. This can affect performace results, with some designs meeting design constraints better with one cost table than others.

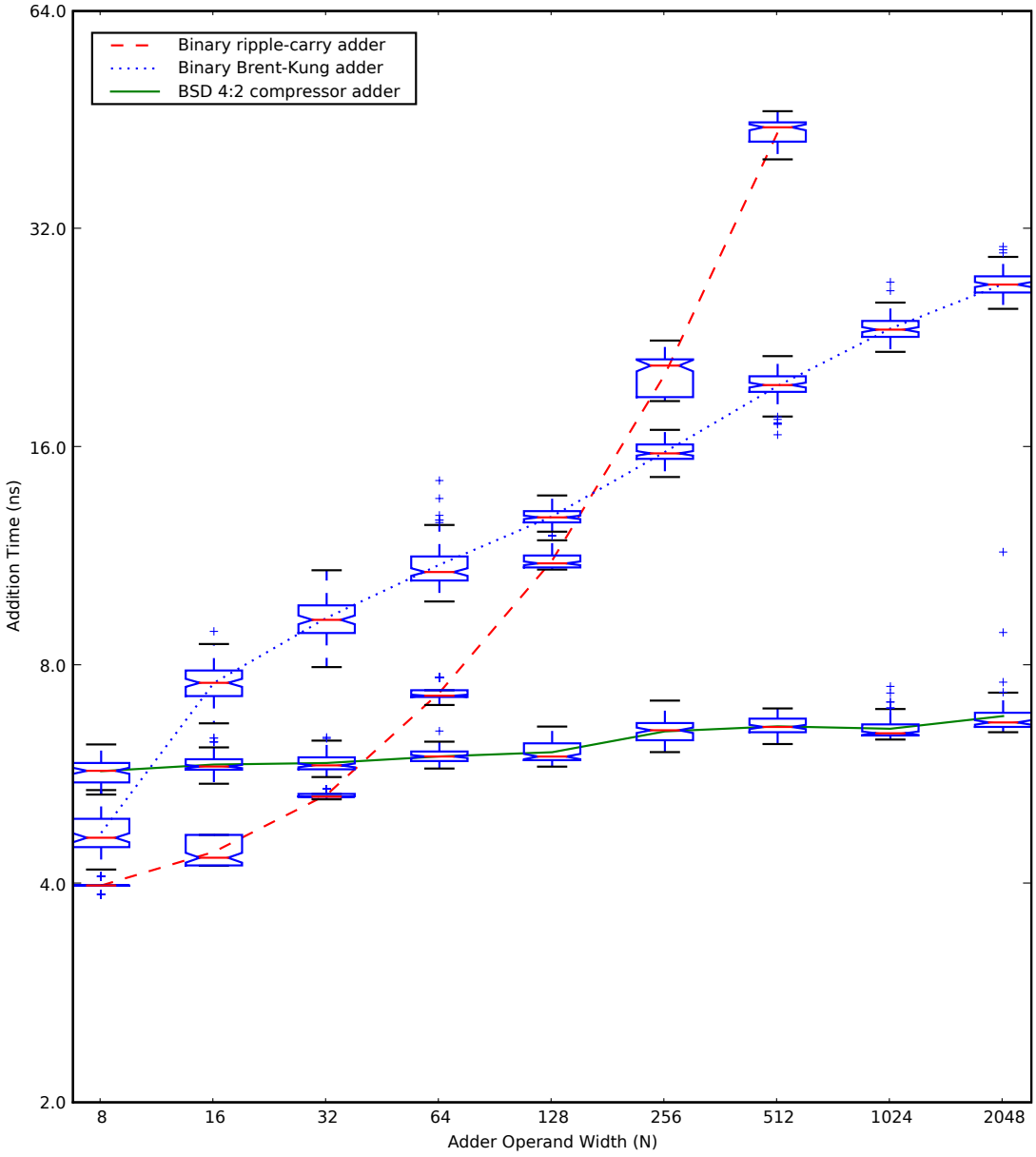


Figure 5.11: A set of box and whisker plots for the performance of adders in a Spartan 3 device. The mean performance points are interpolated by dashed, dotted, and solid lines respectively. The operand widths were varied between 8 and 2048.

Table 5.6: The number of 4-LUTs used for each of the three test adders at various widths in the Spartan 3.

Adder width (N)	Ripple-Carry	BSD Adder	Brent-Kung
8	8	16	29
16	16	32	82
32	32	64	172
64	64	128	406
128	128	256	718
256	256	512	1719
512	512	1024	2626
1024	1024	2048	5797
2048	2048	4096	10281

number system to binary, requires the carry or borrow captured in each symbol to be fully propagated to the end of the word. This is discussed further in Chapter 6. With ripple-carry propagation this takes $O(N)$ time. This can be sped up by using a prefix-tree arrangement like the Brent-Kung adder to give $O(\log N)$ time. Incorporating this into the comparison requires an assumption on the number of additions performed before converting back to binary. Obviously, for only a single addition the time to perform the addition with a redundant number system plus the conversion time will be greater than the time to perform a single binary addition.

5.9 CYCLONE III IMPLEMENTATION

As described in Appendix Section B.1.1, the basic unit of a Cyclone III is the logic element (LE) consisting of a 4-LUT and a register. To implement arithmetic functions efficiently the 4-LUT is fractured into two 3-LUTs, one each to produce the carry-out and sum signals as shown in Figure B.3 (Appendix B).

Since the carry and sum signals are generated in 3-LUTs, a 3:2 compressor of any description can be easily configured straight from Table 5.4 or Table 5.5. However, unlike the Spartan 3, neither the carry-in nor the carry-out signal are exposed to the global interconnect. The fast carry chain must be initiated in the previous LE and terminated by the following LE [Altera Cyclone III, 2008]. This uses two additional LUTs for no functional gain. Despite the two 3-LUTs available in a single LE in arithmetic mode, a standalone 3:2 compressor should be implemented using two 4-LUTs in normal mode, thereby saving a 4-LUT. However, if two or more 3:2 compressors are to be chained, as in Figure 5.9, to produce a group of 4:2 or greater compressors, a different approach

that uses the fast carry logic is more resource efficient.

5.9.1 Two level 4:2 compressors in Cyclone III

The technique described in Section 5.8.3 is again employed to build a chain of 4:2 compressors by using the fast carry logic to diagonally connect 3:2 compressors in adjacent columns and rows. This gives the connections shown in Figure 5.12, where the top LUT initiates the carry chain, the centre two LUTs perform the $\mathbb{PPM}$:$\mathbb{PM}$ and $\mathbb{MMP}$:$\mathbb{MP}$ compressor functions, and the bottom LUT terminates the carry chain. The bottom LE can be combined with the top LE of the next group without interfering with each other. This gives an implementation of approximately 3 LEs for each 4:2 compressor. Structural VHDL is required to achieve this resource usage by routing the output signals through the CARRY_SUM primitive component. This instructs the synthesis engine to use the carry chain [Altera Primitives User Guide, 2007].

To create a larger compressor function of 5 or more inputs at each position an additional level of 3:2 compressors is added for each additional input. The carry chain is used diagonally through the compressor array. This further amortises the extra LE needed to initiate and terminate the carry chain.

5.9.2 BSD adder performance in Cyclone III

A second series of experiments were run to measure the BSD adder performance on a Cyclone III EP3C25U256C6 device. The adder was placed between two registers and its length, N, was varied between 8 and 2048 symbols. It was synthesised using Quartus II version 9.0 SP1. The fitter was run 60 times with different seed values and the maximum propagation delay estimated with the classic timing analyser [Quartus II Handbook, 2009]. The delay of the BSD adder is shown in Figure 5.13. The slight increase in delay as the adder length increases is largely due to interconnect delay as the logic becomes further spread out. The binary ripple-carry and binary Brent-Kung prefix-tree adders were tested for comparison and also plotted in Figure 5.13. The BSD adder and the binary ripple-carry adder have a similar delay at adder lengths of 24 symbols. This is considerably better than the 44 symbol cross-over achieved in the Spartan 3. Again the Brent-Kung only outperforms the ripple-carry at widths greater than 128 bits. The binary ripple-carry adder is limited by the Altera fitter to the LE column height, therefore, only the adders up to 512 bits fit in the selected device.

The LUT usage for each of the three adder designs are given in Table 5.7. The ripple-carry uses the expected N LUTs for an N width adder. The BSD adder as

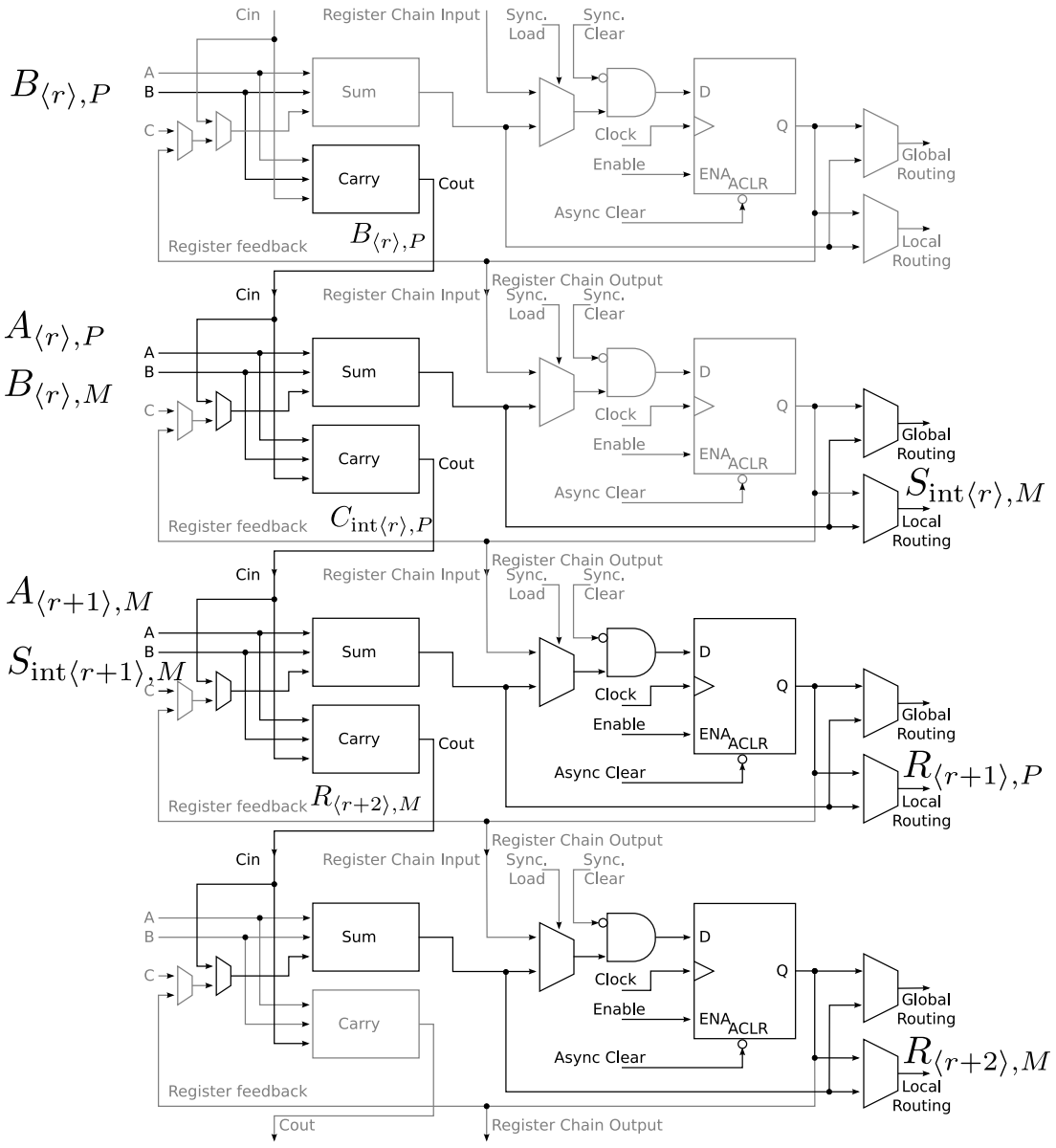


Figure 5.12: A 4:2 compressor function made from a $\mathbb{PPM}$:$\mathbb{PM}$ compressor and a $\mathbb{MMP}$:$\mathbb{MP}$ compressor mapped to four Altera logic elements to implement two half columns of a binary signed digit adder. The dark lines show the logic paths used.

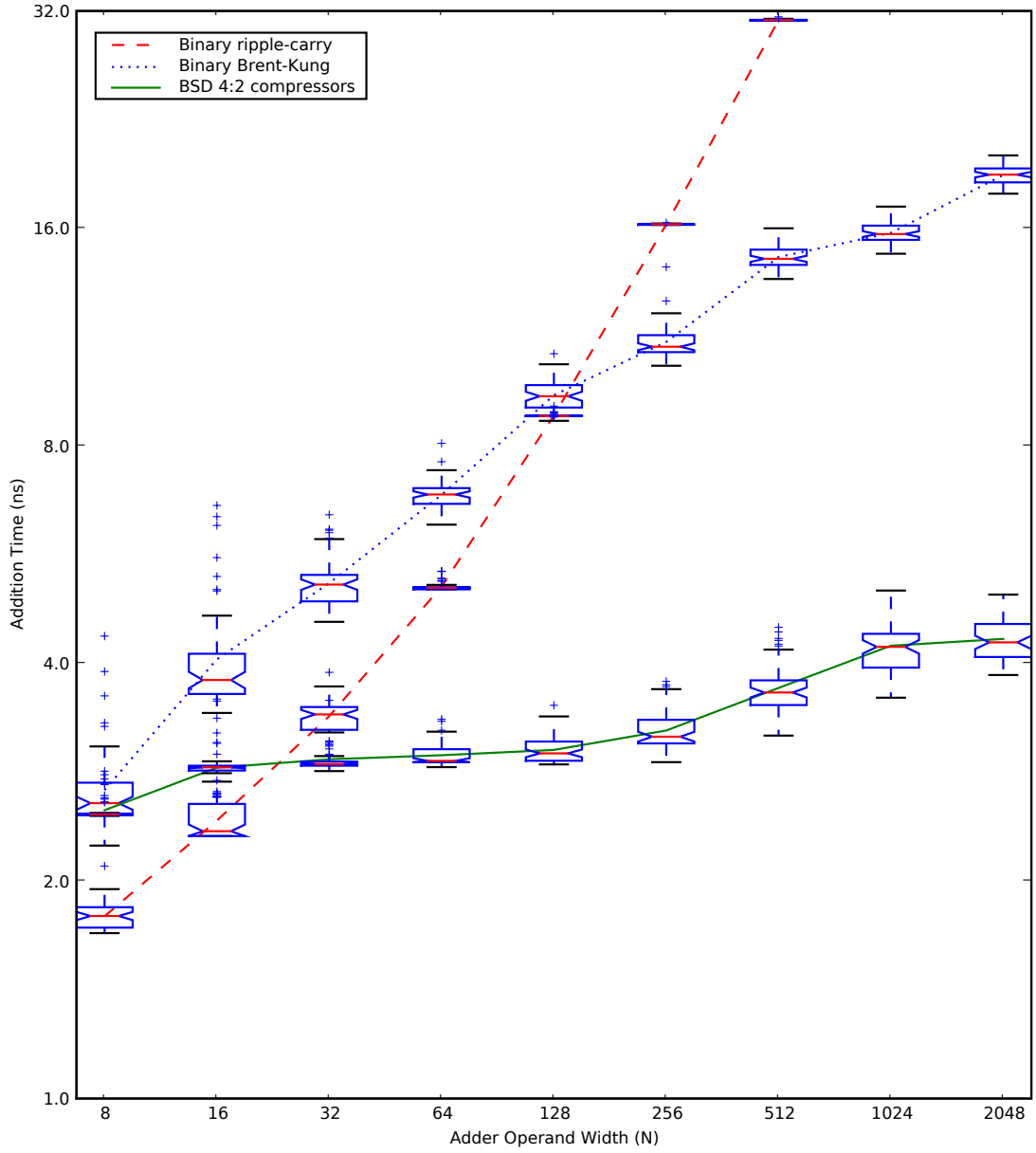


Figure 5.13: A set of box and whisker plots for the performance of adders in a Cyclone III device. The mean performance points are interpolated by dashed, dotted, and solid lines respectively. The operand widths were varied between 8 and 2048.

Table 5.7: The number of 4-LUTs used for each of the three test adders at various widths in the Cyclone III.

Adder width (N)	Ripple-Carry	BSD Adder	Brent-Kung
8	8	25	24
16	16	49	53
32	32	97	113
64	64	193	224
128	128	385	459
256	256	769	926
512	512	1537	1855
1024	-	3073	3718
2048	-	6145	7361

expected uses 3 LUTs per symbol plus one, a total of $3N + 1$ LUTs. The Brent-Kung adder implements more efficiently on the Cyclone 3 than on the Spartan 3, with between $3N$ and $4N$ LUTs over the tested range, $8 \leq N \leq 2048$.

The Cyclone III implementation uses 50% more LUTs than the Spartan 3 implementation since the carry signals are not directly connected to the interconnects. This difference minimises capacitance on the carry signal wire, helping minimise the propagation delay. An architectural change of optionally routing the carry signal to a second LUT could improve resource utilisation for compressor based circuits. This could be achieved by routing the carry signal to the input of another LUT in a neighbouring column. This would require reduced silicon area than routing it through the global interconnect as it requires only a single wire and static multiplexer. This can be envisaged as having a two-dimensional carry chain.

In the comparison it is ignored that the BSD adder is doing twice the work of a binary adder. It is the equivalent of performing two "binary subtractions" with their results merged into a BSD representation. This has parallels to multi-operand addition. Using $\mathbb{PPP} : \mathbb{PP}$ compressors instead gives an implementation for the carry-save redundant number system. It is the basis of Dadda addition trees often used for partial product accumulation in parallel binary multipliers [Dadda, 1977]. In this case there are N constant time additions of N-bit words and one slow ripple-carry addition for conversion at the end. The implementations in Sections 5.8 and 5.9 can equally be applied to multi-operand binary addition providing a fast implementation with the same resource usage as a tree of ripple-carry adders.

5.10 REDUNDANT ADDER RESOURCE USAGE

Using the compressor adder structures, the sizes of adders can be easily estimated given the operand alphabets and their common radix. To add two symbols requires two stages, as discussed in Section 5.6. The first stage performs the initial addition and recoding to signed binary bits, $\mathbb{P}$ and $\mathbb{M}$. The number of LUTs required for this can be estimated from the maximum and minimum sums of the two symbol alphabets. For example, the addition of two operand alphabets $\mathbb{S}_A = \{\overline{1}, 0, 1, 3\}$ and $\mathbb{S}_B = \{\overline{1}, 0, 1, 5\}$ gives the intermediate alphabet $\mathbb{S}_I = \{-2, -1, 0, 1, 2, 3, 4, 5, 6, 8\}$, where the maximum possible symbol sum is 8 and the minimum is -2.

The number of signed bits needed is dependent on both the cardinality of the intermediate alphabet, the values of the symbols, and their spread. The algorithm in Listing 5.1 will determine the number, position, and polarity of the signed bits. This will specify an encoding for each intermediate symbol, although others are possible. Critically for this discussion, the algorithm will determine the minimum number of signed bits, $|\mathbb{B}|$. One cone of logic will be required for each signed bit in $\mathbb{B}$. Assuming each input symbol is binary encoded onto digital signals, each cone of logic has,

$$|\mathbb{I}| = \lceil \log_2(|\mathbb{S}_A|) \rceil + \lceil \log_2(|\mathbb{S}_B|) \rceil, \tag{5.5}$$

inputs. If $|\mathbb{I}| \leq 4$ then each cone can be implemented in a single 4-LUT. Two example encodings are shown in Table 5.8 and Table 5.9. If $|\mathbb{I}| > 4$ then multiple LUTs may be required as shown in Appendix C. This may be reduced through logic minimisation, which is dependent on the digital encoding of the symbols in $\mathbb{S}_A$ and $\mathbb{S}_B$, although it is unlikely.

The second stage of the addition accumulates the signed bits output from the first stage using 3:2 compressors. This will require $(|\mathbb{B}| - 2)$ 3:2 compressors to reduce the signed bits down to two signed bits. One 3:2 compressor can be constructed with one 4-LUT when implemented in a Spartan 3 FPGA as outlined in Section 5.8. When implemented in a Cyclone III the cost is one LUT per compressor plus an extra 4-LUT as in Section 5.9. Thus, the total number of 4-LUTs required is $2(|\mathbb{B}| - 2)$ per position in a Spartan 3 and $2(|\mathbb{B}| - 2) + 1$ in a Cyclone III. Table 5.10 gives a matrix of 4-LUT usage for adders with various symbol alphabets of cardinality four. Table 5.11 gives a similar matrix for subtraction.

Listing 5.1: Algorithm to find the minimum number and polarity of the signed bits used to encode $\mathbb{S}$. The output is an ordered set of integers corresponding to the recoder signed bits; $-1 \rightarrow \mathbb{M}$, $1 \rightarrow \mathbb{P}$, and 0 means no bit required. These are ordered LSB to MSB.

```
1: procedure IsAllZeros(𝕊)
2:     for s ∈ 𝕊 do
3:         if s ≠ 0 then
4:             return False
5:         end if
6:     end for
7:     return True
8: end procedure
9:
10: procedure FindSignedBits(𝕊)
11:     𝔹 = {}                          ▷ Ordered set of signed significant bits, MSB to LSB.
12:     while not IsAllZeros(𝕊) do      ▷ Finish when 𝕊 = {0, ..., 0}.
13:         𝕊_next = {}                 ▷ Alphabet after least significant signed bit removed.
14:         if min(𝕊) = −1 then         ▷ Select signed bit polarity.
15:             b = −1
16:         else:
17:             b = 1
18:         end if
19:         optimise = True             ▷ Flag to remove signed bit if not needed.
20:         for s ∈ 𝕊 do
21:             if s mod 2 = 1 then
22:                 s = s − b
23:                 optimise = False    ▷ Nonzero b is needed.
24:             end if
25:             s = s//2                ▷ Integer divide (round down after division).
26:             𝕊_next = 𝕊_next ∩ {s}
27:         end for
28:         if optimise = True then
29:             b = 0                   ▷ Remove signed bit - make a place holder.
30:         end if
31:         𝔹 = {b} ∩ 𝔹                 ▷ prepend b to MSB.
32:         𝕊 = 𝕊_next
33:     end while
34:     return 𝔹        ▷ Return ordered set of signed significant bits, MSB to LSB.
35: end procedure
```

Table 5.8: An example signed bit encoding for the intermediate sum alphabet $|\mathbb{S}_I| = \{\overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4\} = \{\overline{1}, 0, 1, 3\} + \{\overline{3}, \overline{1}, 0, 1\}$, $\mathbb{B} = \{1, \overline{1}, 1, 1\}$

$s \in \mathbb{S}_I$	=	$b_{\langle 3 \rangle} \in \mathbb{P}$	$b_{\langle 2 \rangle} \in \mathbb{M}$	$b_{\langle 1 \rangle} \in \mathbb{P}$	$b_{\langle 0 \rangle} \in \mathbb{P}$
$\overline{4}$	=	0	$\overline{1}$	0	0
$\overline{3}$	=	0	$\overline{1}$	0	1
$\overline{2}$	=	0	$\overline{1}$	1	0
$\overline{1}$	=	0	$\overline{1}$	1	1
0	=	0	0	0	0
1	=	0	0	0	1
2	=	0	0	1	0
3	=	0	0	1	1
4	=	1	$\overline{1}$	0	0

Table 5.9: Example signed bit encoding for the intermediate sum alphabet $|\mathbb{S}_I| = \{\overline{2}, \overline{1}, 0, 1, 2, 3, 4, 6\} = \{\overline{1}, 0, 1, 3\} + \{\overline{1}, 0, 1, 3\}$, $\mathbb{B} = \{1, 1, \overline{1}, 1\}$

$s \in \mathbb{S}_I$	=	$b_{\langle 3 \rangle} \in \mathbb{P}$	$b_{\langle 2 \rangle} \in \mathbb{P}$	$b_{\langle 1 \rangle} \in \mathbb{M}$	$b_{\langle 0 \rangle} \in \mathbb{P}$
$\overline{2}$	=	0	0	$\overline{1}$	0
$\overline{1}$	=	0	0	$\overline{1}$	1
0	=	0	0	0	0
1	=	0	0	0	1
2	=	0	1	$\overline{1}$	0
3	=	0	1	0	$\overline{1}$
4	=	0	1	0	0
6	=	1	0	$\overline{1}$	0

Table 5.10: Number of Spartan 3 4-LUTs required per symbol position to implement a constant time adder for two radix-2 number systems with four symbol alphabets. Cyclone III usage is given in parentheses.

+	$\{\overline{1}, 0, 1, 3\}$	$\{\overline{1}, 0, 1, 5\}$	$\{\overline{1}, 0, 1, 7\}$	$\{\overline{3}, 0, 1, 3\}$	$\{\overline{3}, 0, 1, 5\}$	$\{\overline{3}, 0, 1, 7\}$
$\{\overline{1}, 0, 1, 3\}$	6(7)	6(7)	6(7)	6(7)	6(7)	6(7)
$\{\overline{1}, 0, 1, 5\}$		6(7)	6(7)	6(7)	6(7)	8(9)
$\{\overline{1}, 0, 1, 7\}$			8(9)	6(7)	8(9)	8(9)
$\{\overline{3}, 0, 1, 3\}$				6(7)	8(9)	8(9)
$\{\overline{3}, 0, 1, 5\}$					8(9)	8(9)
$\{\overline{3}, 0, 1, 7\}$						8(9)

Table 5.11: Number of Spartan 3 4-LUTs required per symbol position to implement a constant time subtractor for two radix-2 number systems with four symbol alphabets. Cyclone III usage is given in parentheses. Subtraction order is column - row.

−	$\{\overline{1},0,1,3\}$	$\{\overline{1},0,1,5\}$	$\{\overline{1},0,1,7\}$	$\{\overline{3},0,1,3\}$	$\{\overline{3},0,1,5\}$	$\{\overline{3},0,1,7\}$
$\{\overline{1},0,1,3\}$	6(7)	6(7)	6(7)	6(7)	6(7)	8(9)
$\{\overline{1},0,1,5\}$	6(7)	6(7)	8(9)	6(7)	6(7)	8(9)
$\{\overline{1},0,1,7\}$	6(7)	6(7)	8(9)	8(9)	8(9)	8(9)
$\{\overline{3},0,1,3\}$	6(7)	6(7)	6(7)	6(7)	8(9)	8(9)
$\{\overline{3},0,1,5\}$	6(7)	8(9)	8(9)	6(7)	8(9)	8(9)
$\{\overline{3},0,1,7\}$	6(7)	8(9)	8(9)	8(9)	8(9)	8(9)

5.10.1 Optimisations

For every signed bit that can be avoided it saves two 4-LUTs per position, one to generate the signed bit and one for the 3:2 compressor. Notice that in the addition of $\{\overline{1},0,1,3\}+\{\overline{1},0,1,3\}$ shown in Table 5.9 the intermediate sum of 6 is the only one that uses a nonzero value for the signed bit $b_{\langle 3\rangle}$. If the arithmetic algorithm can avoid the addition of two 3-symbols then the adder can be two 4-LUTs cheaper and one 3:2 compressor delay faster. Similarly, in the addition of $\{\overline{3},\overline{1},0,1\}+\{\overline{1},0,1,3\}$ shown in Table 5.8, the intermediate symbol 4 is the only symbol using $b_{\langle 3\rangle}$ for a nonzero bit, so avoiding the addition of symbols 3 and 1 will also give the saving. In Section 9.2 the implementation of a dot-product provides a mechanism to avoid the addition of two symbols. For example, the order of addition of partial products is adjusted to avoid the $3+3$ case.

5.11 SUMMARY

Adders that allow their result to be in a redundant number system such as BSD can achieve constant time addition, irrespective of the word length. The general structure used is layers of sequential 3:2 compressors. This avoids the carry propagation in the ripple-carry adder usually employed for arithmetic functions in FPGAs. This in turn gives constant addition time irrespective of word length, which was confirmed experimentally with implementations on both the Spartan 3 and Cyclone III FPGA. When compared to the standard FPGA binary ripple-carry addition, the BSD adder implementation is faster when the operand widths are greater than 24 on the Cyclone III, and 44 on the Spartan 3.

The resource-efficient implementation of fast redundant adders has been shown in the Spartan 3 and Cyclone III low-cost FPGAs. The special addition circuitry associated with each LUT of the Spartan 3 FPGA has been repurposed to implement general 3:2 compressors by employing a structural hardware description. This uses half the logic resources that are usually required with a behavioural hardware description. A standalone 3:2 compressor is only possible with the bottom LUT of a slice since this is where the carry-in signal can be initiated. In the Cyclone III, the arithmetic mode of the logic elements splits the LUT into a user definable 3:2 compressor. However, initialising the carry chain and terminating it both require the use of an additional LUT making singular 3:2 compressor resource inefficient. To improve this, a modification is proposed to optionally route the carry signal to the logic element in the next column. This would create a versatile two dimensional carry network.

The regular layers of sequential 3:2 compressors are usually required to build redundant number system adders. The basic BSD adder uses 4:2 compressors constructed using two layered 3:2 compressors. Utilising the internal carry signal, a single Spartan 3 slice is configured into two halves of a 4:2 compressor, making use of the top LUT in the slice as well. Chaining these compressor blocks builds the adder circuit. Similarly the Cyclone III logic elements are configured as 3:2 compressors and the internal carry chain is used to connect the halves of the 4:2 compressors. This makes their configuration resource-efficient as the initialisation and termination of the carry signal in adjacent compressor functions can use the same LE. Other redundant number systems with irregular symbol alphabets can be summed by applying a recoding layer on top of the 3:2 compressor functions. This is implemented in 4-LUTs.

Chapter 6

CONVERSION AND COMPARISON

A digital signal processing (DSP) subsystem implemented using redundant number systems requires an interface to other system components. Ideally, a system could directly communicate to digital output devices such as digital to analogue converters in a redundant number system. Regardless, a unique representation (or analogue voltage) will need to be derived from the redundant representations at some point where the DSP system interfaces with the real world. Usually this will be through a binary representation. Methods of performing the conversion to and from binary are presented in this chapter. The conversion from binary can be a simple casting from binary bits to equivalent redundant symbols. It can also be a conversion to a specific representation with say minimum Hamming weight.

To use the minimum Hamming weight property effectively a fast method is required to find the minimum weight representations. The serial minimum Hamming weight encoding methods presented in Sections 3.1 and 3.2 are well suited to software implementations or area conscious hardware designs. However, they cannot convert a number fast enough to satisfy the throughput of a parallel multiplier. A fully parallel encoding method has been devised for converting a binary representation to a minimum Hamming weight representation. Implementations for binary signed digit (BSD) and $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$ are presented in Section 6.1.

The conversion from a redundant number system back to binary is a more involved process since there are many redundant representations that must be converted to the same binary representation. As a consequence of the carry-propagation-free addition in Chapter 5 where the carry symbols have be captured within the redundant result representation, the captured carries must now be allowed to propagate from the least significant symbol (LSS) to the most significant symbol (MSS). Therefore, the conversion process is linearly dependent on the word length, like ripple-carry addition. This conversion is a slow process when compared to the constant time addition of Section 5.4. Section 6.2 presents methods to achieve this many-to-one conversion.

Comparing two values is an important operation often used for decision making in algorithms. Comparison has a ternary result that is usually determined by subtracting one value from the other and interpreting the result's sign and magnitude. The subtraction can be executed quickly using a constant time adder, but the redundant result complicates the detection of the sign as there are multiple representations for the same value and no sign bit is generated. Section 6.3 presents a circuit to find, in $O(\log(N))$ time, the most significant nonzero-symbol that indicates the sign of a BSD representation. The same circuit will return a zero if the representation has zero magnitude. This is extended to a comparison operation for two redundant operands in Section 6.4.

6.1 PARALLEL MINIMUM HAMMING WEIGHT ENCODING

The minimum Hamming weight encoding algorithms of Sections 3.1 and 3.2 are presented as finite state machines that serially recode a non-redundant representation. These may be used to recode a binary multiplier to obtain a minimum weight representation. However, the finite state machines only output a single symbol per clock cycle. A serial multiplier, such as that presented in Section 7.3.1, consumes more than one symbol per clock cycle as it skips zero-symbols searching for the next nonzero-symbol. A parallel multiplier, such as that presented in Section 7.3.2, consumes all symbols at once. Therefore, the multiplier words must be precomputed, recoding only a single symbol per clock cycle is insufficient.

This section develops a specific implementation for a BSD parallel minimum Hamming weight encoder. It converts a binary representation to a minimum Hamming weight BSD representation with all symbols encoded in parallel. Note that this does not give a canonical signed digit (CSD) representation[1]. The bit pattern 011_2 may be recoded to either $10\overline{1}_2$ or 011_2 depending on where in the word it occurs. The first logic layer given in Section 6.1.1, recodes the binary input word into BSD using a self-fulfilling assumption. The second layer given in Section 6.1.2, applies an adjustment to remedy any errors the assumption caused. This results in a minimum Hamming weight BSD representation. To further reduce the Hamming weight the BSD representation can be encoded to a $\{\overline{1}, 0, 1, 3\}$ symbol alphabet with slight modification to layers 1 and 2 and the addition of a third layer as described in Section 6.1.3.

[1] see Section 2.4.1

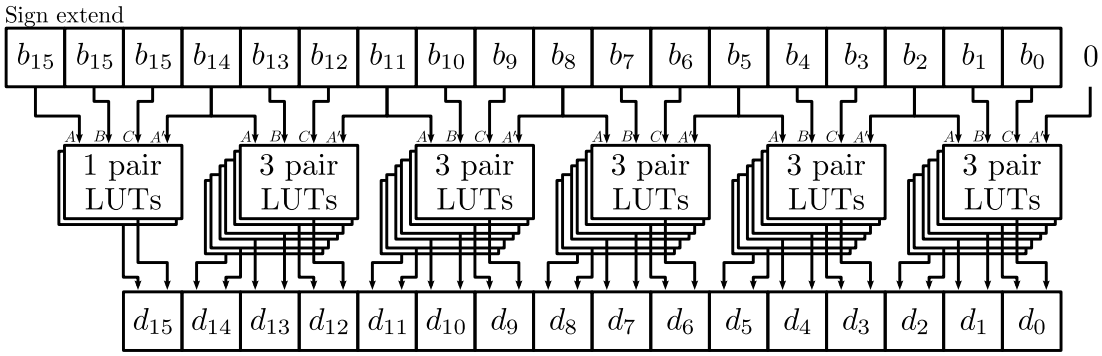


Figure 6.1: Layer 1 of the parallel recoding algorithm, showing how the binary word is sectioned into slices of 3 bits. The inputs to the look-up tables (LUTs) are the slice's bits plus the highest bit from the previous slice.

6.1.1 Layer 1 – Binary to BSD

The first layer in the parallel binary to minimum Hamming weight BSD converter takes as its input a binary word. This binary input word is sliced into groups of three bits (A_1, B_1, C_1). These three bits are transformed into BSD symbols in their corresponding positions. It also peeks at the input bit (A'_1) in the position immediately preceding the slice as shown in Figure 6.1; A'_1 defaults to 0 in the lowest slice. This creates groups of four bits (A_1, B_1, C_1, A'_1) with the top three being recoded to A_2, B_2, C_2 as shown in Table 6.1. The groups of 4-bit are chosen to match the 4-input lookup-tables (4-LUTs) of a Spartan and Cyclone FPGA.

The table is formed by noticing that recoding to CSD replaces all strings of two or more 1-bits by a string of 0-symbols, with a $\overline{1}$-symbol in the least significant position and a 1-symbol in the position above the most significant 1-bit. That is, the patterns $01\cdots1_2$ is replaced by $10\cdots\overline{1}_2$. Depending on where a slice occurs in a string of 1-bits the encoding logic may,

1. Initiate a replacement zero string with a $\overline{1}$-symbol.
2. Continue the zero string with 0-symbols.
3. Terminate a chain with a 1-symbol.
4. A combination thereof.

When two strings of 1-bits are separated by a single 0-bit, for example, 01110111_2. The terminating 1-symbol from the less significant string is included in the following string

Table 6.1: Binary to minimum Hamming weight BSD parallel recoder layer 1 truth table. Rows designated by a † are alternative recodings to the $\{\overline{1}, 0, 1, 3\}$ alphabet and should be used instead of the row above.

	Input Group				Output Group			Chain Description
	A_1	B_1	C_1	A'_1	A_2	B_2	C_2	Action (right to left)
	0	0	0	0	0	0	0	Copy down
	0	0	0	1	0	0	1	Terminate chain
	0	0	1	0	0	0	1	Copy down
	0	0	1	1	0	1	0	Terminate chain
	0	1	0	0	0	1	0	Copy down
	0	1	0	1	0	1	$\overline{1}$	Terminate and copy
†	0	1	0	1	0	0	3	Terminate and combine
	0	1	1	0	1	0	$\overline{1}$	Initiate and terminate
†	0	1	1	0	0	0	3	Combine to 3
	0	1	1	1	1	0	0	Terminate chain
	1	0	0	0	$\overline{1}$	0	0	Initiate chain
	1	0	0	1	$\overline{1}$	0	1	Terminate and initiate
	1	0	1	0	$\overline{1}$	0	1	Copy down and initiate
	1	0	1	1	0	$\overline{1}$	0	Chain continue-skip
	1	1	0	0	0	$\overline{1}$	0	Initiate chain
	1	1	0	1	0	0	$\overline{1}$	Chain continue-skip
	1	1	1	0	0	0	$\overline{1}$	Initiate chain
	1	1	1	1	0	0	0	Chain continue

of 1-symbols, for example, it becomes $0111100\overline{1}_2$ and then the more significant string of 1-bits is recoded to become $1000\overline{1}00\overline{1}_2$.

Peeking at the highest bit of the slice below, A'_1, lets the table determine if it should be initialising or continuing a zero-string in output C_2. The table assumes that if $A'_1 = 1$, then a string of 1-bits reduction chain should be occurring. To fulfil this assumption, if the highest bit of the slice, $A_1 = 1$, then a chain should be continued out of the slice or initialised within the slice.

With the $\{\overline{1}, 0, 1, 3\}$ symbol alphabet the string of two 1-bits can be replaced by a 3-symbol ($011 \rightarrow 003$). This provides a further Hamming weight reduction. Within Table 6.1 the two rows marked with a † are repeated, they implement this replacement and are labelled with the action 'combine', meaning combine two 1-bits to a single 3-symbol.

6.1.2 Layer 2 – Incorrect boundary assumption correction

The assumption made on the highest bit of the previous slice, A'_1, may in some cases generate a recoding across the slice boundary that is not minimum Hamming weight. These boundary errors are fixed in the second layer. There are three error conditions:

1. Terminate-initiate boundary error. If a recoding chain is terminated at symbol $A'_2(= 1)$ and a new chain initiated at symbol $C_2(= \overline{1})$ then this $\overline{1}1$ pattern should be $0\overline{1}$.

2. Initiate-terminate boundary error. In layer 1 the assumption of when a chain was in progress is made on the A'_1. To make this assumption true a chain may be incorrectly initiated at A'_2 and terminated immediately at C_2 giving the pattern $1\overline{1}$ which should have been 01 (a 'copy down').

3. Terminate-copy boundary error (note that this applies to the $\{\overline{1}, 0, 1, 3\}$ recoding only). If a recoding chain is terminated at symbol $A'_2(= 1)$ and symbol $C_2(= 1)$ is copied down in layer 1, then this 11 pattern should be 03.

These boundary conditions and their modifications are summarised in Table 6.2. The logic straddles the boundary of the three bit slices as shown in Figure 6.2. It is across this boundary that the non-CSD recoding to pattern 011_2 can occur. Note that this is still a legal minimum Hamming weight recoding for BSD.

Table 6.2: Boundary error recoding modification table, checking and fixing errors made by the layer 1 assumption across the boundary of three bit slices. The row designated by a † is an alternative recoding used for the $\{\overline{1}, 0, 1, 3\}$ alphabet.

	Input		Output		Boundary Error
	C_2	A'_2	C_3	A'_3	(right to left)
	$\overline{1}$	1	0	$\overline{1}$	Terminate and initiate
	1	$\overline{1}$	0	1	Initiate and terminate
	else		C_2	A'_2	None
†	1	1	0	3	Terminate and copy

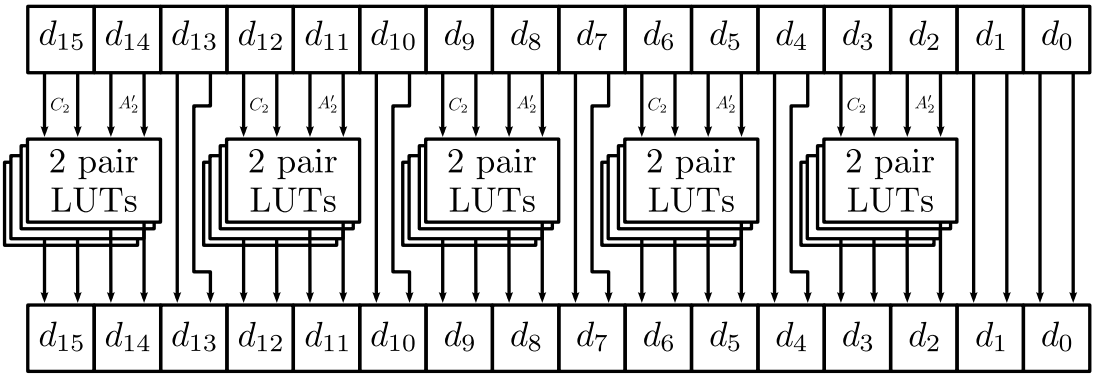


Figure 6.2: Logic arrangement of layer 2 for the parallel recoding of a 16 symbol word.

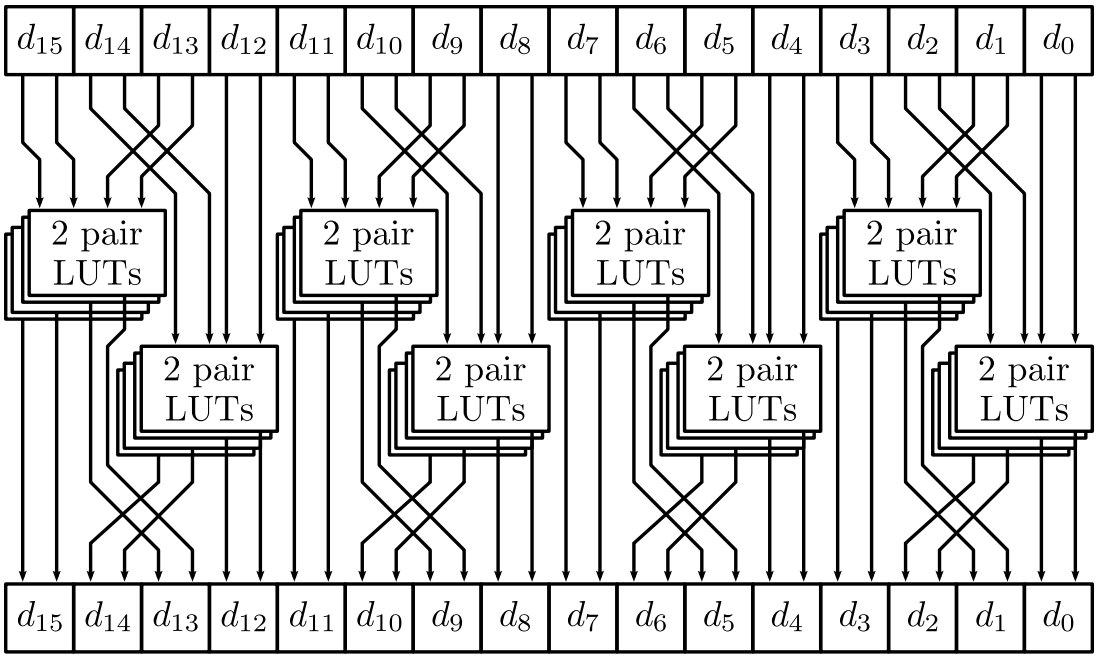


Figure 6.3: Logic organisation for layer 3 of the parallel minimum Hamming weight encoder.

6.1.3 Layer 3 – Four minus one recoding

The output of layer 2 is a minimum Hamming weight BSD representation. However, when recoding to the $\{\overline{1}, 0, 1, 3\}$ alphabet, a final case must be corrected. The value 3 can be expressed in several ways with the $\{\overline{1}, 0, 1, 3\}$ symbol alphabet. These are namely 011_2, 003_2, and $10\overline{1}_2$. When recoding for minimum Hamming weight the 003 representation should be used since it has a Hamming weight of one. The first two layers replace the 011 pattern with 003 but can incorrectly generate the $10\overline{1}$ pattern when:

1. Layer 1 places a $\overline{1}$ symbol in slice position C_2 and then layer 2 corrects the A_2 slice position to be a 1 symbol.

2. Layer 1 places a $\overline{1}$ symbol in slice position B_2 and then layer 2 corrects the C_2 position in the next slice to be a 1 symbol.

This third layer simply looks for the sequence $1 \times \overline{1}$ starting in slice positions B or C and replaces it with 0×3, where $\times$ is a "don't care" condition. The layer 3 logic is interleaved, looking at two symbols two positions apart as shown in Figure 6.3.

6.2 DECODING REDUNDANT REPRESENTATIONS TO THEIR NON-REDUNDANT REPRESENTATIONS

At the completion of a calculation using a redundant number system, the result will likely be in a redundant representation. The result must be decoded to a non-redundant number system such as binary to interface with other devices. Decoding a value's redundant representations to the single non-redundant representation requires a many-to-one mapping. As introduced in (2.5) and Section 2.3, each value has on average $O(|\mathbb{S}|^N/\beta^N)$ redundant representations. Encoding is usually a one-to-one mapping from a non-redundant representation to a specifically structured redundant representation such as one with minimum Hamming weight. See the encoders of Sections 3.1, 3.2, and 6.1. In contrast, the decoding problem cannot assume a nice structured format for the input redundant representation. It is likely unknown and varying, the same value may be represented many different ways at different times. To complicate matters further, the input symbols at the least significant end of the word may have an influence on the decoded symbols at the most significant end of the output word. This is illustrated by the decoding of the value negative one from a BSD representation to two's complement binary,

$$0000000\overline{1}_2 \Longrightarrow 11111111_2,$$

where the $\overline{1}$ symbol in the least significant position influenced all bits in the output two's complement binary word, including the most significant.

The output number system of the adders presented in Section 5.6 are in radix-2 with either the BSD alphabet ($\{\overline{1}, 0, 1\}$), carry-save alphabet ($\{0, 1, 2\}$), or borrow-save alphabet ($\{\overline{2}, \overline{1}, 0\}$). The corresponding non-redundant number system for these radix-2 systems is binary. Section 6.2.1 presents the decoding of BSD to binary, with notes on carry-save and borrow-save which are similar. Sections 6.2.2 and 6.2.3 describe two methods of decoding the $\{\overline{1}, 0, 1, 3\}$, $\beta = 2$ number system to binary.

6.2.1 Decoding BSD to two's complement binary

The conversion process from BSD back to binary can be captured in a Mealy finite state machine with two states, as shown in Figure 6.4. It determines which of three transitions to take from the BSD symbols at each position and outputs a binary bit for every transition taken. The state machine starts in the P state and stays there copying each symbol to the output until a $\overline{1}$-symbol is found. When a $\overline{1}$-symbol is found it transitions to the N state and begins outputting 1-bits for input 0-symbols, and 0-bits for input $\overline{1}$-symbols until a terminating 1-symbol is found. It then outputs a 0-symbol and transitions back to the P state.

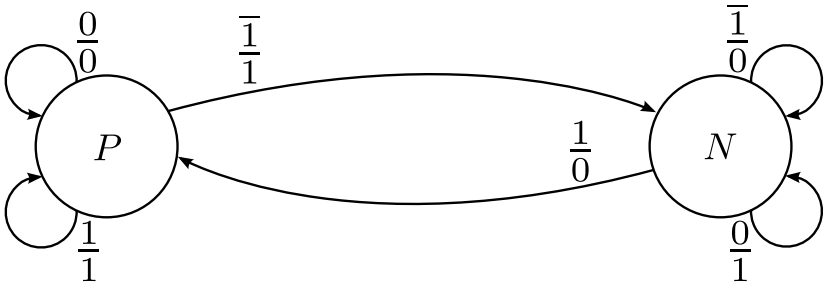


Figure 6.4: Mealy finite state machine for conversion of redundant BSD representations to their corresponding unique binary representation.

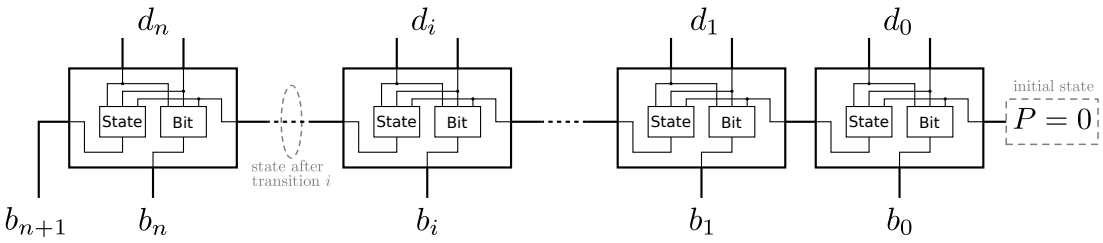


Figure 6.5: Implementation of the BSD to binary conversion state machine. The output bit $b_{\langle i \rangle}$ and the next state are calculated from the current state and input symbol $d_{\langle i \rangle}$.

This state machine can be rolled out and implemented with two three input lookup tables (3-LUTs) per position, as shown in Figure 6.5. One LUT determines the output bit from the input bit and the current state. The other LUT determines the next state from the input symbol and the current state. Using the logic elements of an Altera Cyclone FPGA in arithmetic mode allows the two 3-LUTs to be implemented in a single 4-LUT. The state signal is transported between adjacent logic elements using the fast carry routing. Using this routing decreases the propagation delay of the state signal, although it must still travel the full length of the word.

If the BSD symbols are encoded using a $\mathbb{P} = \{0, 1\}$ signal and a $\mathbb{M} = \{\overline{1}, 0\}$ signal, like those emitted by the adder of Section 5.6, then the BSD word can be split into two unsigned binary words. The positive word has all the $\mathbb{P}$ bits and the negative word has all the $\mathbb{M}$ bits, in their same positions. To decode into a two's complement signed result the negative word is subtracted from the positive word using a binary subtractor [Kuninobu et al., 1987a]. Similarly, for carry-save encoded on two $\mathbb{P}$ signals the conversion is an addition and for borrow-save encoded on two $\mathbb{M}$ signals the conversion is an addition followed by a negation.

6.2.2 Decoding $\{\overline{1}, 0, 1, 3\}$ to two's complement binary with a finite state machine

The serial conversion from the redundant number system $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$ to binary can be described using the Mealy finite state machine shown in Figure 6.6. This state machine extends the BSD state machine of Figure 6.4, including two extra states for decoding the additional 3-symbol:

1. 'Single 3' is transitioned to when a 3-symbol is encountered, or when the value remaining to be encoded is 3.

2. '3 String' is transitioned to when consecutive 3-symbols are encountered. When traversing to this state the value remaining to be encoded is 4; a carry two places ahead in the binary word.

The state machine starts in state P decoding from the least to most significant symbol. It traverses the input representation, emitting one binary bit for each transition. An example of decoding a redundant representation to a binary word is shown in Table 6.3.

Unrolling the state machine creates a cascade of logic that operates in a word parallel, symbol serial fashion, as shown in Figure 6.7. Three 4-LUTs are required for each symbol position, each takes two state state signals and two signals for the in redundant symbol, d_i, as inputs. The next state is encoded on two output bits and the third output gives the binary bit (b_i). Although the input symbols are provided in parallel, the conversion is not complete until the state signals have propagated the entire length of the logic cascade.

6.2.3 Decoding $\{\overline{1}, 0, 1, 3\}$ to binary using binary adders

A possibly faster and more resource efficient method in an FPGA is to split the symbols into three separate binary words and combine them using an adder and a subtractor. The symbol alphabet $\{\overline{1}, 0, 1, 3\}$ with the signal encoding of Table 6.4 is used as an example for this discussion but the same principle can be applied to other symbol alphabets.

The first step in decoding is to split the symbol encoding up into a collection of signed bits. These bits have either positive or negative value and a power of 2 weighting. The value of each symbol can be constructed by adding a combination of these bits. The symbol alphabet $\{\overline{1}, 0, 1, 3\}$ can be mapped to signed bits as shown in Table 6.4.

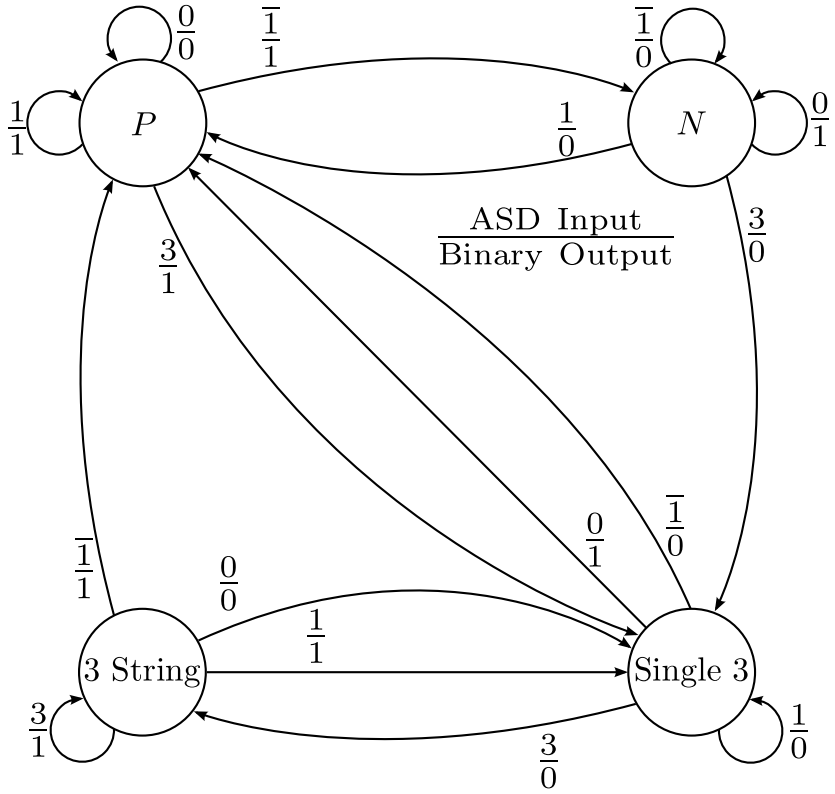


Figure 6.6: Mealy finite state machine for conversion of redundant $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$ representations to their corresponding unique binary representation.

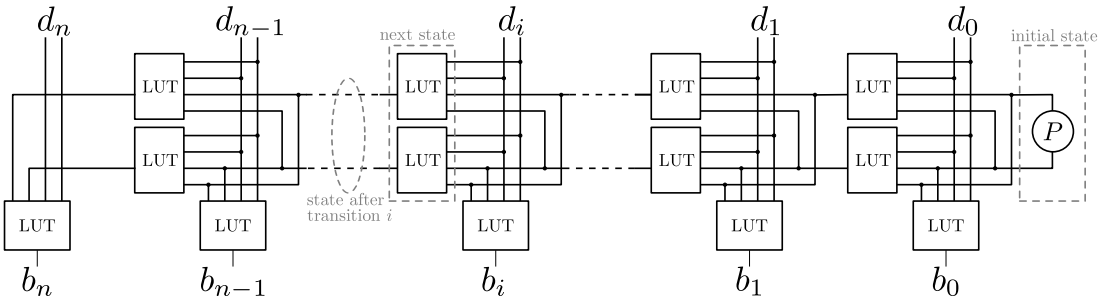


Figure 6.7: The Mealy state machine unrolled to create a cascade of logic.

Table 6.3: Example of using the state machine of Figure 6.6 to decode the representation $01\overline{1}3\overline{11}33\overline{11}1010333\overline{10}\overline{1}_2$ with decimal value 313131_{10} to the equivalent binary representation of 01001100011100101011_2.

Step	Current State	$\{\overline{1}, 0, 1, 3\}$ input	Transition	Output	Next State
0	P	$\overline{1}$	P→N	1	N
1	N	0	N→N	1	N
2	N	1	N→P	0	P
3	P	3	P→Single 3	1	Single 3
4	Single 3	3	Single 3→3 String	0	3 String
5	3 String	3	3 String→3 String	1	3 String
6	3 String	0	3 String→Single 3	0	Single 3
7	Single 3	1	Single 3→3 String	0	Single 3
8	Single 3	0	Single 3→P	1	P
9	P	1	P→P	1	P
10	P	$\overline{1}$	P→N	1	N
11	N	$\overline{1}$	N→N	0	N
12	N	3	N→Single 3	0	Single 3
13	Single 3	3	Single 3→3 String	0	3 String
14	3 String	$\overline{1}$	3 String→P	1	P
15	P	$\overline{1}$	P→N	1	N
16	N	3	N→Single 3	0	Single 3
17	Single 3	$\overline{1}$	Single 3→P	0	P
18	P	1	P→P	1	P
19	P	0	P→P	0	P

Table 6.4: Two encodings of the symbols $\{\overline{1}, 0, 1, 3\}$. The first on two digital signals P and M and the second set of signed bits, derived from binary sign-magnitude, appropriate for recoding to binary using adders. The $p_{\langle 1 \rangle}$ is a positive bit with weight 2 giving it the possible values $\{0, 2\}$.

Symbol	Compact Encoding		Expanded Encoding (signed bits)		
	$\mathbb{P}$	$\mathbb{M}$	$p_{\langle 1 \rangle}$	$p_{\langle 0 \rangle}$	$n_{\langle 0 \rangle}$
$\overline{1}$	0	1	0	0	1
0	0	0	0	0	0
1	1	0	0	1	0
3	1	1	1	1	0

Furthermore, the signed bits can be expressed as Boolean equations in $\mathbb{P}$ and $\mathbb{M}$. For this example,

$$p_{\langle i+1 \rangle} = \mathbb{P}\mathbb{M}, \tag{6.1}$$

$$p_{\langle i+0 \rangle} = \mathbb{P}, \tag{6.2}$$

$$n_{\langle i+0 \rangle} = \overline{\mathbb{P}}\mathbb{M}. \tag{6.3}$$

Converting each symbol to these signed bits and grouping the bits of the same sign and weighting gives several unsigned binary words. For example, the $n_{\langle i+0 \rangle}$-bits are grouped to give a binary word with negative magnitude. Each of the unsigned binary words is normalised in weight by shifting the higher weighted words left, i.e., the binary word made from the $p_{\langle i+1 \rangle}$-bits are shifted left one position. Then the negative words are subtracted and the positive words are added with binary ripple-carry adders to give the final binary two's complement result.

This method uses the fast carry chains in the FPGA architectures, therefore, it should perform better than the straight state machine method. In a Xilinx Spartan 3 FPGA the equations (6.1) though (6.3) can be implemented within the same LUT as the addition, reducing the example to two 4-LUTs per position. In an Altera Cyclone III FPGA putting the logic element in arithmetic mode reduces the input signals to two plus the carry-in. Thus (6.1) would need to be implemented in a separate LUT before addition, maintaining a 3 LUT per position resource usage.

6.2.4 Delay and area

Decoding to binary is a necessarily slow operation due to the large number of representations with the same value. Using a large LUT to perform the mapping would use

too many resources to implement, since generating the most significant bit (MSB) of the output requires inputs from the entire redundant word. It would also require the enumeration of all possible input representations. Decoding by the serial methods presented in Sections 6.2.1 to 6.2.3 is linearly proportional to the word length, $O(N)$. This is at least as slow as a ripple-carry binary adder. For this reason, the conversion should be performed at a late stage to amortise the delay cost over as many fast redundant addition computations as possible. The techniques used to increase binary addition performance, such as carry look-ahead [Yen et al., 1992], may be applied to speed up conversion at the expense of increased resource usage, see appendix A. However, as indicated by the Brent-Kung adder tested in Sections 5.8.4 and 5.9.2, this strategy consumes many resources and does not provide a speed advantage until word lengths of greater than 128 symbols.

6.3 SIGN AND ZERO DETECTION

Many algorithms require knowledge of a variable's sign or when it has zero magnitude. Determining the sign of a binary two's complement representation simply requires checking the sign bit, i.e., the MSB. However, determining if the word is zero involves checking the entire representation contains 0-bits. Redundant representations often do not use a sign bit, the negativity of its value is determined by the sign of its symbols, see Section 2.2. When all the magnitudes of the symbols in the alphabet are less than the radix, such as in BSD, the sign of the number is the sign of the most significant non-zero-symbol (MSNZS). However, if the symbol alphabet has symbols with a magnitude greater than the radix, as the 3-symbol is for radix-2, then these symbols can influence the sign of the representation from less significant positions, overriding the MSNZS.

The families of representations listed in Table 6.5 all have positive values but a $\overline{1}$-symbol at the MSNZS. Note that for all these representations, if decoded to binary with the state machine of Figure 6.6, they will take a transition to the starting state (P) with their most significant $\overline{1}$-symbol. As a consequence, the MSNZS is insufficient to determine the sign of the representation and all symbol positions must be checked.

Two methods can be used to determine the sign of a redundant representation:

1. Decode the representation to binary two's complement and inspect the sign bit. Some resources can be saved by only producing the most significant bit (sign bit). For example, with the decoder design of Section 6.2.2, only the state signals need to be generated, not the output symbols. This method has $O(N)$ resource usage and $O(N)$ delay.

Table 6.5: Families of positive $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$ representations with $\overline{1}$ as the most significant non-zero-symbol.

$0 \ldots 0\overline{1}3\mathrm{X} \ldots \mathrm{X}$
$0 \ldots 0\overline{1}13\mathrm{X} \ldots \mathrm{X}$
$0 \ldots 0\overline{1}1 \ldots 13\mathrm{X} \ldots \mathrm{X}$
$0 \ldots 0\overline{1}03 \ldots 3\mathrm{X} \ldots \mathrm{X}$
$0 \ldots 0\overline{1}103 \ldots 3\mathrm{X} \ldots \mathrm{X}$
$0 \ldots 0\overline{1}1 \ldots 103 \ldots 3\mathrm{X} \ldots \mathrm{X}$

2. Convert the representation to a representation with all symbol magnitudes less than the radix. For example, convert $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$ to a BSD representation, which, by definition (having no 3-symbol) does not have the problematic representations shown in Table 6.5. This conversion can be done quickly in $O(1)$ time. The BSD word is then inspected with a binary tree structure to find the MSNZS. This method has a resource usage of $O(N)$ and speed of $O(\log(N))$.

6.3.1 Constant time conversion to BSD

A redundant representation with symbols of magnitude greater than the radix can be converted to a BSD word quickly and in constant time irrespective of the word length. This is achieved using the logic of the constant time adders in Section 5.6, with one operand set to a constant zero. For $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$, the logic minimisation from addition with zero gives the two 4-LUT per position implementation shown in Figure 6.8. If the symbol in the lower position $a_{\langle i-1 \rangle} = 3$ then current position should expect a carry-in. The LUTs generate a negative bit in the same position ($n_{\langle i \rangle}$) and a positive carry bit into the next position ($p_{\langle i+1 \rangle}$). The output bits in the same position form a BSD output symbol ($s_{\langle i \rangle}$).

6.3.2 Sign and zero detection of a BSD representation

After conversion to BSD, the sign of the MSNZS accurately represents the sign of the value. The BSD representation is searched to find the MSNZS using a tree structure that reduces the word to a single symbol as shown in Figure 6.9. If a node's left input is nonzero then it passes to the level the value from its left branch, otherwise it passes its right branch value. In this way the MSNZS is propagated to the root of the tree. The circuit will additionally report when the representation is zero. This occurs when all symbols are zero; the right branch of each node is selected, propagating the zero at

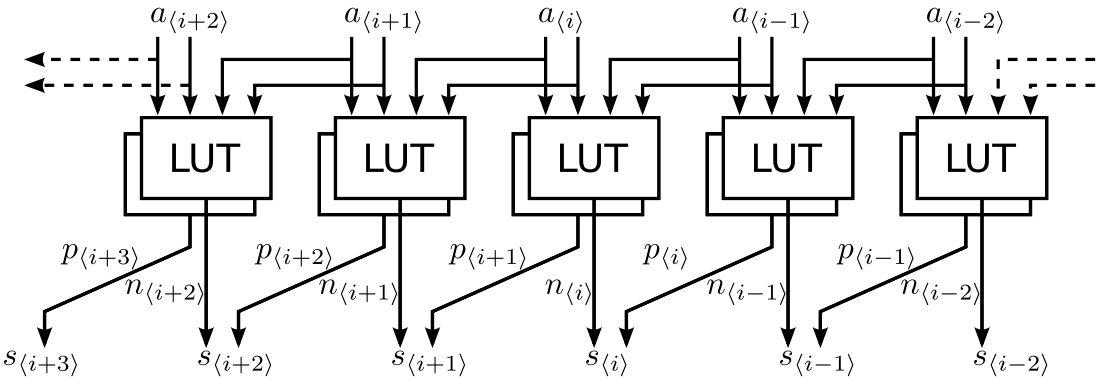


Figure 6.8: Design for a $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$,$\beta = 2$ to BSD converter utilising two LUTs per symbol position.

the least significant position to the root.

Each node in the tree can be implemented using two 3-LUTs as multiplexers, with two inputs from the left symbol and one from the right symbol. Only one right symbol signal is required per LUT since it is not used to determine the selection, it is only used to copy to the LUT output. Both of the left inputs are required to determine the selection. Therefore, the LUT usage of the binary tree structure is proportional to the word length, $O(N)$ and the delay through the tree is proportional to its depth, $O(\log_2(N))$.

Some resources maybe saved in an ASIC design if the BSD symbols are encoded on two bits with one for sign and the other for magnitude. This encoding is shown in Table 6.6. This encoding simplifies the multiplexer for the left and right magnitude bits L_{mag} and R_{mag} to an OR gate,

$$\begin{aligned} F_{\text{mag}} &= L_{\text{mag}}L_{\text{mag}} + \overline{L_{\text{mag}}}R_{\text{mag}}, \\ &= L_{\text{mag}} + R_{\text{mag}}. \end{aligned} \tag{6.4}$$

The sign bits L_{sign} and R_{sign} still requires a multiplexer selected on L_{mag},

$$F_{\text{mag}} = L_{\text{mag}}L_{\text{sign}} + \overline{L_{\text{mag}}}R_{\text{sign}}. \tag{6.5}$$

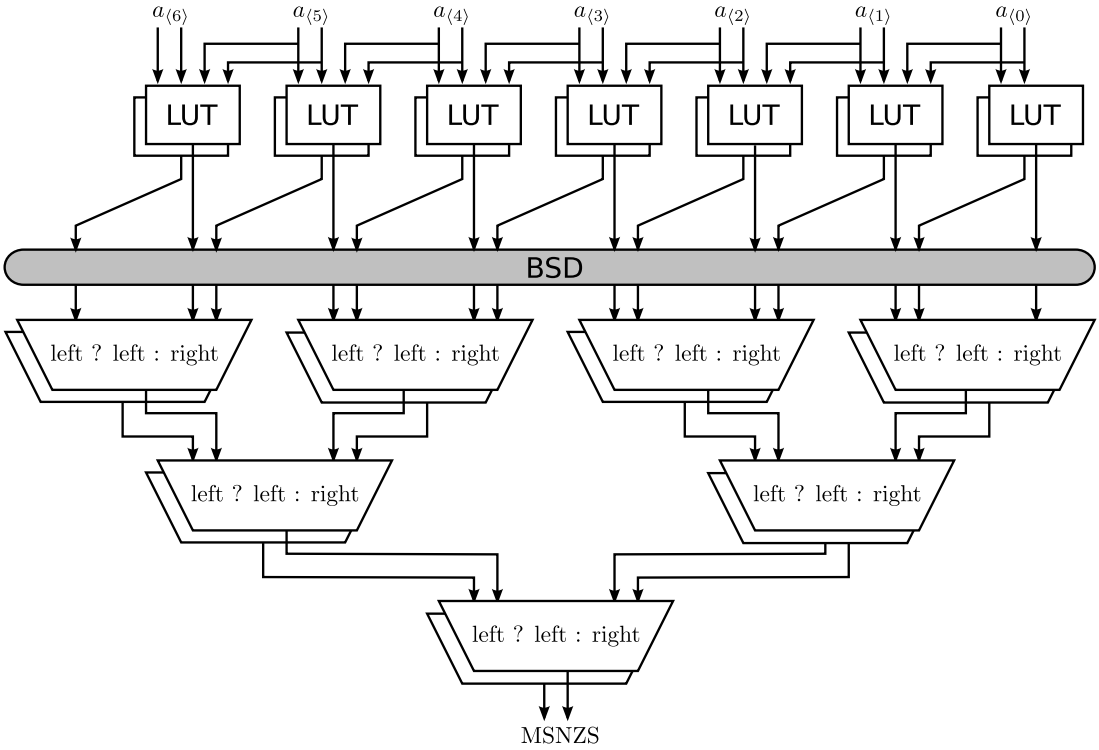


Figure 6.9: Sign and zero detection circuit for a seven symbol $\mathbb{S} = \{\overline{1}, 0, 1, 3\}, \beta = 2$ word employing a conversion to an eight symbol BSD representation and a tree structure to find the most significant nonzero-symbol (MSNZS).

Table 6.6: BSD symbol encoded on sign and magnitude signals.

$\text{sign}_{\langle i\rangle}$	$\text{mag}_{\langle i\rangle}$	$\rightarrow$	$s_{\langle i\rangle}$
$\overline{1}$	1	$\rightarrow$	$\overline{1}$
$\overline{1}$	0	$\rightarrow$	0
0	0	$\rightarrow$	0
0	1	$\rightarrow$	1

6.4 COMPARISON OPERATION

The comparison of two representations, A and B, has three different possible outcomes,

1. A less than B,
2. A equal to B, or
3. A greater than B.

To determine the result, B is subtracted from A and the sign of the result is inspected using the method in Section 6.3.2. If the subtraction result is positive then A is greater than B, if it is negative then A is less than B, and if it is zero then A is equal to B.

The sign detection tree of Figure 6.9 outputs one of three symbols, $\{\overline{1}, 0, 1\}$ corresponding to the comparison relationships $\{<, =, >\}$. There is a fourth unused state available on the two output signals. It may be tempting to use this fourth state for a relation such as much greater than (symbol $\ggg$) or much less than (symbol $\lll$). This does not work for most redundant representations, especially if negative symbols are used. Developing this idea further exposes the error.

The designer might choose the threshold between $>$ and $\ggg$, or $<$ and $\lll$ to be a power of two magnitude difference for simplicity. At the position corresponding to that power of two the sign detection tree is modified to return the fourth symbol $\ggg(\lll)$ to the root instead of $>$ $(<)$, indicating the MSNZS was in a position greater than or equal to that position, hence a 'large' magnitude value. However, this would be incorrect as the position of the MSNZS does not indicate magnitude in a BSD representation. Consider a comparison of two numbers that differ in value by only one. Depending on the order of subtraction this gives the result of positive one or negative one. In BSD these could be expressed as $1\overline{11}\cdots\overline{1}_2$ and $\overline{1}11\cdots1_2$ respectively. The MSNZS does give the correct sign, but its position falsely indicates a large magnitude.

6.5 SUMMARY

It is usually satisfactory to perform a one-to-one conversion from a non-redundant representation such as binary to a redundant representation. It can be a simple casting or an algorithm to generate a pseudo-redundant representation with a particular property such as minimum Hamming weight. The minimum Hamming weight encoding state machines of Chapter 3 can perform this conversion in a time linearly proportional to the word length. This may not be fast enough to satisfy a parallel multiplier. Therefore,

a fully parallel implementation was developed for BSD and $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$,$\beta = 2$ that showed this conversion can be accomplished in constant time.

The conversion from a redundant representation back to a unique non-redundant representation is a many-to-one mapping. All possible redundant representations of a value must decode to the same unique non-redundant representation for that value. As discussed in Chapter 5, a constant time adder uses a redundant output representation to avoid propagating a carry. The carry is captured in the redundant representation. To decode the representation requires that these captured carries be allowed to propagate, possibly the full length of the representation. As a consequence, the decoding takes a time proportional to the word length. Therefore, decoding should be left to as late a stage as possible in the DSP algorithm so that the decoding delay can be amortised over many constant time additions. The implementation of these decoders can take the form of a finite state machine or a series of non-redundant adders as shown for the BSD and $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$,$\beta = 2$ number systems.

No explicit sign information is kept with the representation of a number as it is for signed binary representations, two's complement or otherwise. This makes determining the sign of a number difficult, especially if the absolute value of any symbol is greater than or equal the radix. When the maximum symbol magnitude is less than the radix, as it is for BSD, the sign of the number is the sign of the MSNZS. To find this requires a tree search of the representation taking $O(\log N)$ time. It also reveals if the magnitude of the representation is zero. To determine if a non-redundant representation is zero also takes $O(\log N)$ time, so redundant representation are not at a disadvantage.

Finally, the magnitude of a redundant representation is *not* indicated by the position of the MSNZS. This was shown for a value of one, where the MSNZS could be in any position.

Chapter 7

SHIFTING: MULTIPLICATION AND DIVISION

In multiplication algorithms, a zero-symbol in the multiplier operand generates a zero partial product. Since this does not change the value of the accumulation, the associated addition can be skipped. The lower minimum Hamming weight of a redundant representation means that there are more zero-symbols and therefore, more additions can be skipped. The multiplication algorithm is discussed in Section 7.1 and methods of exploiting the minimum weight property in Section 7.2. Section 7.3 presents a serial and parallel multiplier architectures. Further exploitation of the minimum Hamming weight property is presented in Chapter 8 where the algorithm is a dot-product consisting of many multiplications.

Division is simple when dividing by a power of the radix. The quotient is simply calculated by shifting all symbols right a number of positions equal to the power. To maintain an integer result, the symbols below the zeroth position, the fractional part, are usually truncated. However, a truncation error occurs when discarding the least significant symbols of a redundant representation. This error is different to that expected with binary arithmetic. An explanation and possible solutions are detailed in Section 7.4.1.

7.1 MULTIPLICATION

Multiplication is a common operation used in many signal processing algorithms. Multiplication of two positional number representations is commonly achieved using the long multiplication algorithm. This algorithm breaks the multiplication up into a sum of many small multiplications called partial products [Mano and Kime, 2004]. A partial product is the multiplicand operand $\mathbf{b} = \{b_{\langle n-1\rangle}, b_{\langle n-2\rangle}, \ldots, b_{\langle 1\rangle}, b_{\langle 0\rangle}\}$, multiplied by the j^{th} symbol ($a_{\langle j\rangle}$) of the multiplier operand $\mathbf{a} = (a_{\langle n-1\rangle}, a_{\langle n-2\rangle}, \ldots, a_{\langle 1\rangle}, a_{\langle 0\rangle})$. Partial products are generated from the combination of two simple multiplication cases;

Table 7.1: The symbol multiplication table for $\{\overline{1}, 0, 1, 5\}$ and $\{-1, 0, 1, 3\}$ giving the partial product alphabet $\{\overline{5}, \overline{3}, \overline{1}, 0, 1, 3, 5, 15\}$.

$\cdot$	$\overline{1}$	0	1	5
$\overline{1}$	1	0	$\overline{1}$	$\overline{5}$
0	0	0	0	0
1	$\overline{1}$	0	1	5
3	$\overline{3}$	0	3	15

1. Multiplication of two symbols. This is easily accomplished with a small table look-up. The table contains the result for all possible pairs of symbols, one selected from each of the two operands' respective symbol alphabets. The alphabets do not need to be the same. An example symbol multiplication table is shown in Table 7.1. If the multiplicand alphabet is binary ($\{0, 1\}$) then the partial product alphabet is that of the other operand, since 0 is the multiplicative absorbing element and 1 is the multiplicative identity.

2. Multiplication of a representation by a power of the radix, i.e., β^k. This is easily accomplished by shifting all symbols in the representation left by k positions, denoted $\ll k$. The k least significant positions are filled with zero symbols.

$$\mathbf{b} \times \beta^k = \mathbf{b} \ll k. \tag{7.1}$$

Therefore, to compute the partial product of $a_{\langle j \rangle}$ times $\mathbf{b}$, each symbol of $\mathbf{b}$ is first multiplied by $a_{\langle j \rangle}$. Then the resulting representation is multiplied by β^j according to the ordinal position of $a_{\langle j \rangle}$,

$$\mathrm{pp}(a_{\langle j \rangle}; \mathbf{b}) = (a_{\langle j \rangle} b_{\langle n-1 \rangle}, a_{\langle j \rangle} b_{\langle n-2 \rangle}, \ldots, a_{\langle j \rangle} b_{\langle 1 \rangle}, a_{\langle j \rangle} b_{\langle 0 \rangle}) \times \beta^j. \tag{7.2}$$

In the simple case where $\mathbf{b}$ is a binary representation, the partial product is the multiplicand $\mathbf{b}$ with all 1-bits replaced by the multiplying symbol $a_{\langle j \rangle}$ and shifted left j positions.

To compute the full multiplication of $\mathbf{a} \times \mathbf{b}$, a partial product is generated for each symbol in $\mathbf{a}$ and then the partial products are summed,

$$\mathbf{a} \times \mathbf{b} = \sum_{j=0}^{N-1} \mathrm{pp}(a_{\langle j \rangle}; \mathbf{b}). \tag{7.3}$$

$\mathbf{b} =$								1	1	1	0	1	0	1	1	$= 235_{10}$
$\times\mathbf{a} =$								0	$\overline{1}$	0	0	1	0	0	3	$= -53_{10}$
$\mathrm{pp}(3 \cdot 2^0; 11101011) =$								3	3	3	0	3	0	3	3	$= 705_{10}$
$\mathrm{pp}(0 \cdot 2^1; 11101011) =$							0	0	0	0	0	0	0	0		$= 0_{10}$
$\mathrm{pp}(0 \cdot 2^2; 11101011) =$						0	0	0	0	0	0	0	0			$= 0_{10}$
$\mathrm{pp}(1 \cdot 2^3; 11101011) =$					1	1	1	0	1	0	1	1				$= 1880_{10}$
$\mathrm{pp}(0 \cdot 2^4; 11101011) =$				0	0	0	0	0	0	0	0					$= 0_{10}$
$\mathrm{pp}(0 \cdot 2^5; 11101011) =$			0	0	0	0	0	0	0	0						$= 0_{10}$
$\mathrm{pp}(\overline{1} \cdot 2^6; 11101011) =$		$\overline{1}$	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	0	$\overline{1}$	$\overline{1}$							$= -15040_{10}$
$+\mathrm{pp}(0 \cdot 2^7; 11101011) =$	0	0	0	0	0	0	0	0								$= 0_{10}$
$=$	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	1	0	1	2	3	3	1	4	0	3	3	$= -12445_{10}$
(BSD result) $=$	0	$\overline{1}$	$\overline{1}$	0	0	0	0	$\overline{1}$	$\overline{1}$	0	1	1	1	$\overline{1}$	$\overline{1}$	$= -12445_{10}$

Figure 7.1: An example long multiplication of an eight bit binary representation for 235_{10} and a minimum Hamming weight redundant representation for -53_{10} using the symbol alphabet $\{\overline{1}, 0, 1, 3\}$ and radix 2.

Expanding (7.3) to a tabular form shows each partial product,

$$\mathbf{a} \times \mathbf{b} = \begin{array}{|cc|}\hline & \mathbf{a} \times \mathbf{b} \\ \hline & \mathrm{pp}(a_{\langle 0 \rangle}; \mathbf{b}) \\ + & \mathrm{pp}(a_{\langle 1 \rangle}; \mathbf{b}) \\ + & \vdots \\ + & \mathrm{pp}(a_{\langle N-2 \rangle}; \mathbf{b}) \\ + & \mathrm{pp}(a_{\langle N-1 \rangle}; \mathbf{b}) \\ \hline \end{array} . \tag{7.4}$$

This form will be used later in Section 8.1 to visualise a dot-product.

An example of an eight by eight symbol multiplication is shown in Figure 7.1. The multiplicand operand $\mathbf{b}$ is an eight bit unsigned binary representation and the multiplier operand $\mathbf{a}$ is a eight symbol minimum Hamming weight representation with the symbol alphabet $\{\overline{1}, 0, 1, 3\}$. Note that the partial products are the $\mathbf{b}$ operand with all 1-bits changed to $\mathbf{a}_{\langle j \rangle}$ and shifted left j positions.

7.2 ELIMINATING PARTIAL PRODUCTS

A multiplier's performance is limited by the number of additions that must be performed. If some partial products can be eliminated then the multiplication time is reduced as fewer additions are required. To reduce the number of nonzero partial products and hence the number of additions there are two methods:

1. Make the multiplier operand a higher radix representation by grouping several symbols and multiplying by their combined values. Consequentially the symbol multiplication table is much larger. The multiplication algorithms of Booth [1951] and its derivatives [Sam and Gupta, 1990] in essence recode the binary two's complement multiplier operand into canonical signed digit (CSD) and then group symbols in pairs. Due to the CSD recoding one symbol in the pair will be zero, making a radix-4 representation with symbol set $\{-2, -1, 0, 1, 2\}$. The number of additions is half the width of the original binary word for a grouping of two (radix-4) and a third for a grouping of three bits (radix-8).

2. Find a minimum Hamming weight representation of the multiplier operand. Every zero-symbol in the multiplier operand generates a zero partial product. The addition of zero partial products can be ignored, reducing the multiplication time [Koc and Johnson, 1994]. The zero partial products are easily identified ahead of time by the zero-symbols in the multiplier operand. This is the primary motivation for minimising the Hamming weight of representations.

Removing the zero partial products from the example of Figure 7.1, reveals that only two additions are required for this multiplication, as shown in Figure 7.2. If all zero partial products can be avoided, the worst case number of additions is $\lceil N/2 \rceil$ for a multiplier employing the BSD/CSD number system. The average case depends on the average minimum Hamming weight of the multiplier symbol alphabet. BSD has an average minimum Hamming weight of $\frac{1}{3}N + \frac{1}{9}$ [Hartley, 1996]. The average and worst case minimum Hamming weights of other number systems are presented and discussed in Sections 10.4 and 10.5 respectively.

7.3 REDUNDANT NUMBER SYSTEM HARDWARE MULTIPLIER ARCHITECTURES

In the usual construction of a multiplier the left shifts of the partial products are fixed. To maximise the gain from ignoring zero partial products the redundant number system multiplier architecture needs a mechanism to skip to the nonzero partial products. This

$\mathbf{b} =$							1	1	1	0	1	0	1	1	$= 235_{10}$
$\times\mathbf{a} =$							0	$\overline{1}$	0	0	1	0	0	3	$= -53_{10}$
$\text{pp}(3 \cdot 2^0; 11101011) =$							3	3	3	0	3	0	3	3	$= 705_{10}$
$\text{pp}(1 \cdot 2^3; 11101011) =$				1	1	1	0	1	0	1	1				$= 1880_{10}$
$+\text{pp}(\overline{1} \cdot 2^6; 11101011) =$		$\overline{1}$	$\overline{1}$	$\overline{1}$	0	$\overline{1}$	0	$\overline{1}$	$\overline{1}$						$= -15040_{10}$
$=$	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	1	0	1	2	3	3	1	4	0	3	3	$= -12445_{10}$
(BSD result) $=$	0	$\overline{1}$	$\overline{1}$	0	0	0	0	$\overline{1}$	$\overline{1}$	0	1	1	1	$\overline{1}$	$\overline{1}$	$= -12445_{10}$

Figure 7.2: An example multiplication with the zero partial products removed, requiring only two additions to compute the result.

would usually be a variable shift executed between subsequent partial products. There are two types of architectures that are traditionally used for multiplication:

1. A serial multiplier that computes a single partial product and adds it to a running accumulation each clock cycle. This requires a small amount of logic but has a long delay of N clock cycles.
2. A parallel multiplier that generates all partial products at once and adds them together in a single clock cycle. This has a small delay but requires a large amount of logic.

7.3.1 Serial multiplier

The serial multiplier architecture performs one partial product accumulation every clock cycle. Many clock cycles are required for each multiplication but comparatively few resources are used [Brown and Vranesic, 2003]. A general serial multiplier structure is shown in Figure 7.3.

The multiplication begins by loading the two operands and zeroing an accumulator. On each clock cycle the serial multiplier performs the following steps:

1. Select the next symbol from the multiplier operand.
2. Shift the accumulation right one position as shown in Figure 7.3. This is equivalent to shifting the multiplicand operand left one position, but more efficient [Chu, 2006].
3. Generate the partial product from the multiplicand and multiplier symbol.

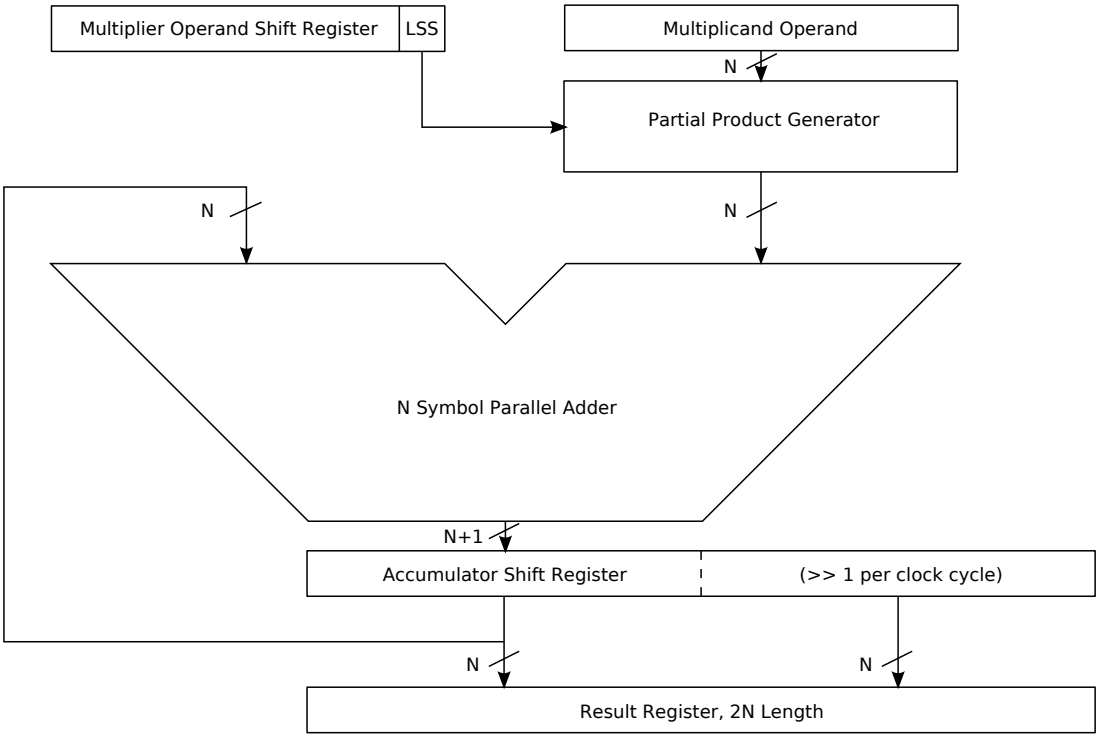


Figure 7.3: A general serial multiplier structure. All registers have the least significant symbol (LSS) on the right. Every clock cycle one partial product is added to the top N bits of the accumulator which is subsequently shifted right, effectively implementing the left shift of the next partial product.

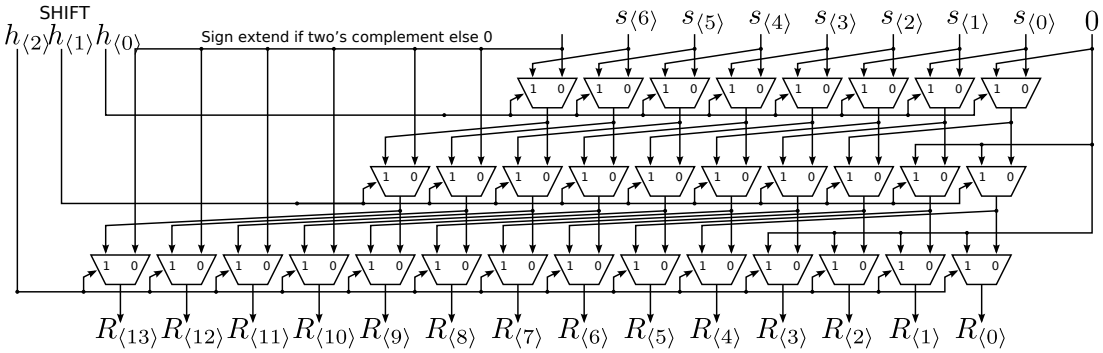


Figure 7.4: Three level selectable shifter that can produce any shift between zero and seven positions in a single clock cycle, given a binary shift value $h_{\langle 2\rangle}h_{\langle 1\rangle}h_{\langle 0\rangle\ 2}$.

4. Add the partial product to the running accumulation of partial products.

When the last multiplier symbol has been used, the multiplication result is read from the accumulator register and stored in the result register. The accumulator can then be zeroed, ready for the next multiplication.

The zero partial products can be skipped using a barrel shifter circuit that can shift a representation a selectable number of positions, $h \in [0, h_{\max}]$, in a single step. In general, a barrel shifter is implemented with multiplexers that select which symbol will move to which position. In an FPGA this would be most efficiently implemented as layers of multiplexers as shown in Figure 7.4. Each layer increases the number of shifts by a factor of two [Chu, 2006].

The variable shift is complicated by the selection of the next symbol from the multiplier operand. The operand must be searched for the next nonzero symbol and its relative position, i.e., its shift. The number system may also specify a minimum shift, $g \geq 1$, if it is known that there are at least $g-1$ zero-symbols between nonzero symbols. This increases the total shift range from $[0, h_{\max}]$ to $[g, g+h_{\max}]$ by pre-shifting (or post-shifting) the representation with a fixed hard-wired shift. For example, CSD with at least one zero between nonzero symbols has a minimum shift of $g = 2$. Therefore, the shifting logic is designed with a static shift of two followed by a variable shift of up to $h_{\max}$. A shifter larger than $g + h$ can be achieved in multiple hops where an intermediate zero partial product is added. For example, consider the design for a CSD serial multiplier with a variable shift of $h \in [0, 7]$ and a static shift of $g = 2$. To achieve a 10 position shift, outside the $\langle 2+7\rangle$ shift range, requires a $\langle 2+6\rangle$ position shift followed by a $\langle 2+0\rangle$ position shift in sequence. Note that using the maximum $h = 7$ variable shift $\langle 2+7\rangle$ followed by the minimum shift $\langle 2+0\rangle$ does not give a total shift of 10. To maintain the two hop maximum range of $\langle 2+7\rangle + \langle 2+7\rangle = 18$ and

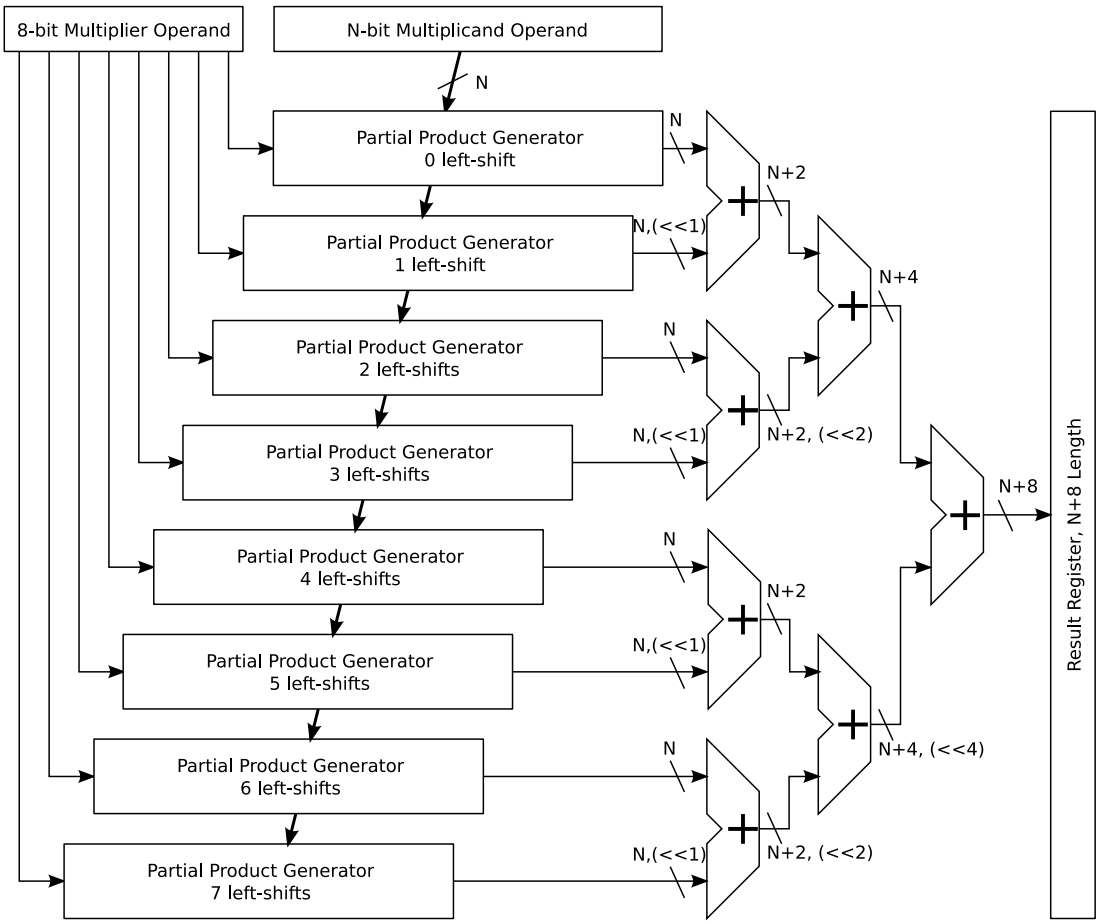


Figure 7.5: A general parallel multiplier structure with an eight symbol multiplier and an N symbol multiplicand.

avoid overshooting a shift of 10, the search for the next nonzero symbol must be in the range $[g, 2(g+h)-1]$ positions higher than the current position.

7.3.2 Parallel multiplier

The parallel multiplier architecture performs one complete multiplication every clock cycle using multiple partial product generating circuits and multiple adders. It is generally faster than the serial multiplier but it uses many more resources. The general parallel multiplier structure is shown in Figure 7.5. A partial product is generated for every position in the multiplier operand, each with a fixed left shift. The partial products are added together either sequentially or with a tree structure as in Figure 7.5.

A partial product R is formed between a symbol $a_{\langle j \rangle}$ and a binary word $\mathbf{b}$ using AND gates as shown in the 'symbol multiplication' half of Figure 7.6. The partial product symbol $R_{\langle i \rangle}$ is equal to $a_{\langle j \rangle}$ if the symbol $b_{\langle i \rangle} = 1$, otherwise $R_{\langle i \rangle} = 0$. A selectable

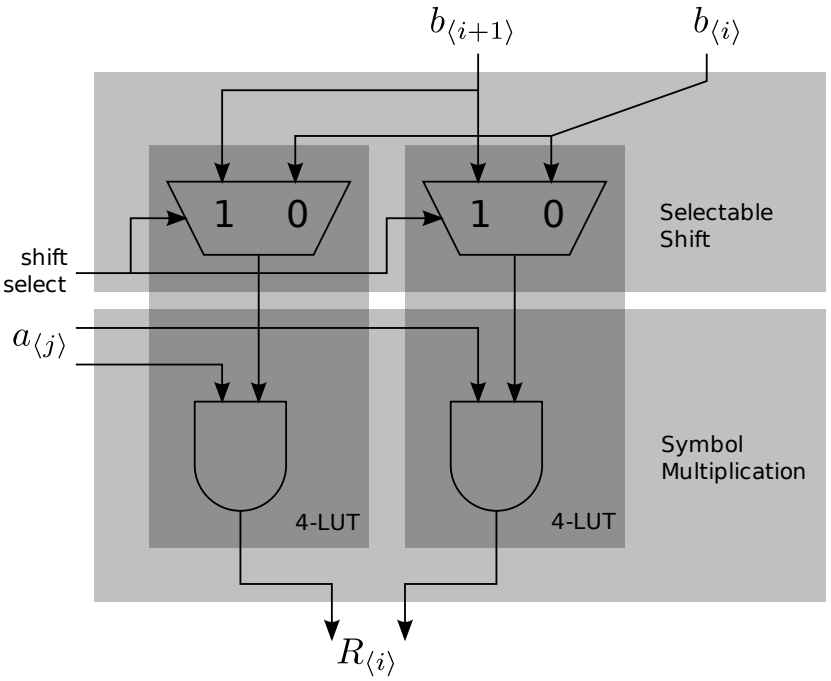


Figure 7.6: A variable shifter that can select either of two shifts, combined with a partial product generator. It is implementable in a single level of logic using two 4-input look-up tables.

shift selects either $\mathbf{b} \times \beta^j$ or $\mathbf{b} \times \beta^{j+1}$ for the current position in order to generate a partial product for a shift of j or $j+1$ respectively. This is shown by the 'selectable shift' half of Figure 7.6. The shift is selected by the position of the nonzero multiplier symbol. In this way the partial product generation logic can be shared by two positions of the multiplier operand, $a_{\langle j\rangle}$ and $a_{\langle j+1\rangle}$. The logic arrangement of Figure 7.6 can be implemented efficiently in two 4-input look up tables (4-LUTs).

The partial product logic of Figure 7.6 can be utilised if the redundant multiplier operand representation a is chosen so that its symbols can be regularly paired so that only one symbol in the pair is nonzero, like in CSD. The variable shift halves the number of adders required, providing greater utilisation of the remaining adders as they add more non-zero partial products, since zero partial products are only generated when both symbols in the pair are zero. For example, CSD has the property that every nonzero symbol is followed by a zero-symbol. Thus adjacent partial products can be paired with the guarantee that only one symbol will ever be nonzero. Therefore, the number of partial product generating circuits and adders can be halved.

This design can be used with a binary multiplicand and an up to 4 symbol alphabet multiplier operand. Implementing it in an FPGA with 4-LUTs employs otherwise unused inputs in a 4-LUT. Therefore, it does not drastically increase the resource usage over a design without the variable shift. There will be a small increase to allow for the

control logic and symbol selection, which is amortised over all positions in the partial product.

7.4 DIVISION, TRUNCATION AND ROUNDING

Division is the inverse operation of multiplication, however, it is often inaccurate in finite precision as the result is truncated at the lower symbols to fit the specified range. Division of two arbitrary numbers is a comparatively expensive operation in digital hardware. The simplest algorithms require subtraction, right shifting and sign detection [Brown and Vranesic, 2003]. Due to its complexity division is expensive to implement as well as slow. Where possible in signal processing algorithms, division is minimised, restricted to divisors of β^x, or completely avoided.

Division by a power of the radix (β^x) is simply a shift of the dividend x places to the right, denoted $\gg x$. For example, to divide representation A by β^x, shift all symbols in A to positions x places lower,

$$B = \frac{A}{\beta^x} \equiv A \gg x, \tag{7.5}$$

where $\gg$ is the right shift operator. The symbols of A are moved to different positions. This can be achieved with hard-wiring if x is a fixed integer. This is equivalent to simply interpreting the ordinates differently,

$$\begin{aligned}
b_{\langle n-1-x\rangle} &\leftarrow a_{\langle n-1\rangle},\\
b_{\langle n-2-x\rangle} &\leftarrow a_{\langle n-2\rangle},\\
\vdots \quad &\vdots \quad \vdots\\
b_{\langle i-x\rangle} &\leftarrow a_{\langle i\rangle},\\
\vdots \quad &\vdots \quad \vdots\\
b_{\langle 0\rangle} &\leftarrow a_{\langle x\rangle},\\
b_{\langle -1\rangle} &\leftarrow a_{\langle x-1\rangle},\\
\vdots \quad &\vdots \quad \vdots\\
b_{\langle -x+1\rangle} &\leftarrow a_{\langle 1\rangle},\\
b_{\langle -x\rangle} &\leftarrow a_{\langle 0\rangle}.
\end{aligned} \tag{7.6}$$

If x is not fixed then a barrel shifter can be used similar to that described for a left shift in Figure 7.4.

For the result B to fit in the same positions as the original divisor A, the lower

x positions, $b_{\langle -1 \rangle}, \ldots, b_{\langle -x \rangle\ \beta}$, are truncated. It is said that these symbols have been shifted out. If the dividend is an integer, i.e., with a lowest positional ordinate of zero, then the result is truncated below the zero-ordinate to also be an integer, losing the fractional part.

Truncation may also be used to reduce the width of the representation [Koren, 2002]. For example, this may be required after a multiplication where the full precision result has a word length equal to the sum of the operand word lengths. The result can be reduced to the same word length as an operand by discarding the LSS until the desired width is reached. Only the most significant positions are kept. In a non-redundant number system the result will always be the same. However, with a redundant number system, a value's representations have nonzero-symbols in different positions. Truncating the LSSs of two different representations may yield different results. Defining and correcting this issue, using the case of right shifting to perform simple division, is discussed in Sections 7.4.1 and 7.4.2.

7.4.1 Single arithmetic right shift

To divide a number by the radix requires all symbols to be shifted to the right one place. To maintain the same integer resolution between the operand and result the LSS is truncated and discarded. In non-redundant arithmetic this effectively rounds the result *down* to the nearest integer. However, this is not always the case when using redundant number representations.

For example, take the number 15 in binary and integer-divide (//) by $\beta = 2$, hence right shift 15 by one ($\gg 1$) and truncate the least significant bit (LSB) to give the 'correct' integer result,

$$15_{10}//2 \overset{\text{binary}}{\longrightarrow} 01111_2 \gg 1 = 00111_2 \overset{\text{dec}}{\longrightarrow} 7_{10}. \tag{7.7}$$

Using the same right shift and truncate operation for the number 15 recoded in minimum Hamming weight BSD results in,

$$15_{10}//2 \overset{\text{BSD}}{\longrightarrow} 1000\overline{1}_2 \gg 1 = 01000_2 \overset{\text{dec}}{\longrightarrow} 8_{10}. \tag{7.8}$$

This result is different to that expected by the non-redundant calculation of (7.7). Consider that the $\overline{1}$-symbol in the least significant position initiates a string of 1-bits in the decoding of BSD to binary. Truncating the $\overline{1}$-symbol from the end of the shifted representation, the string of 1-bits that it initiates in the decoding is also removed, leaving only the terminating 1-symbol. Therefore, the result is one larger than is expected. To fix this, a value of one needs to be subtracted from the result, equivalently

adding a $\overline{1}$-symbol.

Further to the example, another possible representation of the value 15 is 00303_2. Performing the right shift and truncate gives,

$$15_{10}//2 \overset{\{\overline{1},0,1,3\}}{\longrightarrow} 00303_2 \gg 1 = 00030_2 \overset{\text{dec}}{\longrightarrow} 6_{10}. \tag{7.9}$$

This representation gives an unexpected result of 6. In this case a 3-symbol was shifted out. Again consider the decoded binary equivalent of the 3-symbol in the least significant position. It represents two binary 1-bits, in its position and one higher. Therefore, by removing the 3-symbol two 1-bits are removed from the binary equivalent, the higher one unexpectedly. To fix this, a value of one needs to be added to the result.

As a final example case, consider where the representation for 15 has a 1-symbol in the lowest position. The shift right and truncate gives,

$$15_{10}//2 \overset{\{\overline{1},0,1,3\}}{\longrightarrow} 00311_2 \gg 1 = 00031_2 \overset{\text{dec}}{\longrightarrow} 7_{10}. \tag{7.10}$$

The result is as expected and requires no futher action. A 1-symbol decodes to a binary 1-bit, which is correctly removed. If a 0-symbol is the LSS then the number is a multiple of β and it may also be removed without correction; it always decodes to a binary 0-bit.

Also note that right shifting a representation leaves the most significant symbol (MSS) unoccupied. To maintain the sign of two's complement binary representation during right shifting the MSB keeps the same bit as had previously. The sign of a redundant representation is maintained by the most significant nonzero symbol (MSNZS), so right shifting does not affect it. A 0-symbol is always shifted into the most significant position.

To summarise, unlike binary a single arithmetic right shift of a positive redundant representation cannot be achieved solely with wiring. In the case that the dividend's LSS is not in the non-redundant alphabet $\{0, \cdots, \beta - 1\}$, the result is corrected by adding a correction value. The correction addend is found by representing the shifted out symbol value in its non-redundant form, right shifting it one place and truncating. Examples of correction addends are shown in Table 7.2 for a collection of shifted out symbols in radix-2. Commonly, a right shift is followed by an addition, so the correction addend may be added using the carry-in of the following adder.

Truncation is a rounding towards negative infinity. For a negative number to round towards zero requires that a value of one be added before truncating the last position. This is true for all representations including binary.

Table 7.2: Correction addends to correct the result of an integer after being shifted right one place and truncated. The symbol shifted out is the LSB of the dividend. The radix is assumed to be 2.

Shifted out symbol	Correction addend
$\overline{7}$	$\overline{4}$
$\overline{5}$	$\overline{3}$
$\overline{3}$	$\overline{2}$
$\overline{1}$	$\overline{1}$
0	0
1	0
3	1
5	2
7	3

7.4.2 Multiple arithmetic right shift

If shifts of more than one are required then more symbols are shifted out of the representation. The error in the result will still be of the same size as the single shift case, however, it is more difficult to determine the error and correction factor than looking at the last symbol shifted out. The entire string of shifted out symbols must be considered when determining the correction addend.

Consider an N symbol word A with symbols $a_{\langle N-1\rangle}a_{\langle N-2\rangle}\ldots a_{\langle i\rangle}\ldots a_{\langle 1\rangle}a_{\langle 0\rangle}$, that is right shifted by x symbols to give the result $R = A \gg x$. the first shifted out symbol is $r_{\langle -x\rangle} = a_{\langle 0\rangle}$ and $r_{\langle -1\rangle} = a_{\langle x-1\rangle}$ is the last. The truncation of the shifted word to make it an integer leaves the symbols $R = r_{\langle N-1-x\rangle}r_{\langle N-2-x\rangle}\ldots r_{\langle i-x\rangle}\ldots r_{\langle 1\rangle}r_{\langle 0\rangle}$ as the $N - x$ symbol result of integer division by β^x.

To determine the correction addend, the x shifted out symbols $r_{\langle -1\rangle}, \ldots, r_{\langle -x\rangle}$ are decoded to a non-redundant word $B = b_{\langle y-1\rangle}, \ldots, b_{\langle -1\rangle}, \ldots, b_{\langle -x\rangle}$, where y is the number of extra positions needed to represent the value in binary. For example, BSD requires $y = 1$ for a sign bit. Truncating the symbols of B below $b_{\langle 0\rangle}$ gives the correction addend value. If the correction addend is negative, then replace the 1-symbols with $\overline{1}$-symbols.

7.4.2.1 Performance, resources, and accuracy tradeoff

Decoding all the shifted out symbols to determine the correction addend is a slow process compared to the shifting operation that can often be hard-wired. Several

options provide a tradeoff in terms of performance, resources, and accuracy,

1. As described in Section 7.4.2, decode the shifted out symbols into a non-redundant word and use the result to decide on the correction addend. This is accurate, uses resources proportional to the shift, but has a high delay.

2. Ignore the correction symbol if the application can manage the inaccuracy in the shift result, i.e., just use a logical right shift. This has no logic or time delay associated with it, however, there is increased error in the LSS. This manifests itself in DSP algorithms as increased noise. For BSD this noise will be approximately double that of binary, as it rounds the result up sometimes and down sometimes, instead of always down.

3. Guess the correction addend based on the last symbol shifted out. Note that each symbol will not necessarily give the same addend each time. To work out the best guess, the symbol probabilities at the post-shifted position $\langle -1 \rangle$ are needed. From this the best correction addend can be chosen by considering the improvement and detriment to accuracy, in the cases of a correct guess and a false-positive condition. Increased accuracy may also be obtained by looking at more of the shifted out symbols. This is fast and uses few resources, If done correctly it can reduce the error in comparison to option 2.

The inaccuracy may be mitigated by increasing the word width by y. This gives no increase in delay, but potentially causes a large increase in logic as the increased word width cascades through the following arithmetic functions.

Chapter 8

OPTIMISED DOT-PRODUCT EVALUATION

The dot-product is a vector function frequently used in digital signal processing (DSP). It forms the basis of many numerically intensive algorithms, in particular filtering, correlation, and matrix multiplication [Ifeachor and Jervis, 2002]. Therefore, an efficient dot-product in terms of performance, cost and energy is important.

The dot-product of two length L vectors,

$$\mathbf{A} = (\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{L-1})^T$$

and

$$\mathbf{B} = (\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{L-1})^T$$

can be written as,

$$\mathbf{A} \bullet \mathbf{B} = \sum_{i=0}^{L-1} (\mathbf{a}_i \times \mathbf{b}_i)\,, \tag{8.1}$$

where the plain subscript i indexes the coefficients in vectors $\mathbf{A}$ and $\mathbf{B}$. The dot-product result is a single scalar value. Digital filtering uses the dot-product extensively. For example, it is used to calculate each output sample from a finite impulse response (FIR) filter, where vector $\mathbf{A}$ is the FIR coefficients describing the filter response and $\mathbf{B}$ contains the L latest digital samples of the input signal being filtered.

A typical implementation of a dot-product employs a multiply accumulate unit (MAC). It consists of a multiplier and an accumulator that sums the sequential pairwise multiplication results. The MAC function is used L consecutive times, once for each $(\mathbf{a}_i, \mathbf{b}_i)$ pair, to generate the dot-product result as a direct implementation of (8.1).

If the elements of vector $\mathbf{A}$ have a word length of N then a parallel multiplier requires N partial product generating circuits with fixed shifts and an addition tree, see Section 7.3.2. Decomposing the L multiplications in (8.1) into their $L \times N$ constituent partial products illustrates the dot-product's dependency on addition and partial product

generation,

$$\mathbf{A} \bullet \mathbf{B} = \sum_{i=0}^{L-1} \sum_{j=0}^{N-1} \mathrm{pp}(a_{i\langle j\rangle}, \mathbf{b}_i), \tag{8.2}$$

where the subscript within angle braces $\langle j \rangle$ indexes the symbol within the representation of coefficient $\mathbf{a}_i$. The partial product for the jth position of $\mathbf{a}_i$ is,

$$\mathrm{pp}(a_{i\langle j\rangle}, \mathbf{b}_i) = a_{i\langle j\rangle} \mathbf{b}_i \beta^j, \tag{8.3}$$

$$\mathrm{pp}(a_{i\langle j\rangle}, \mathbf{b}_i) = (a_{i\langle j\rangle} b_{i\langle N-1\rangle},\ a_{i\langle j\rangle} b_{i\langle N-2\rangle},\ \ldots\ ,\ a_{i\langle j\rangle} b_{i\langle 0\rangle}) \ll j. \tag{8.4}$$

The β^j factor is usually implemented as a fixed shift of j places to the left, written $\ll j$.

As presented in Chapter 3, the zero partial products resulting from the multiplication with a zero-symbol ($a_{i\langle j\rangle} = 0$) can be excluded from the accumulation without affecting the result. Section 8.1 uses this result in a novel way to eliminate the zero partial products and pack partial products from other multiplications in their place. This reduces the computational load on the dot-product hardware; allowing it to complete earlier.

Section 8.2 applies redundant number representations to the dot-product to improve the packing and hardware performance. However, this does not give optimal packing; methods for improvement are suggested in Section 8.3. A novel method that uses the redundancy in representation is described as a combinatorial optimisation problem in Section 8.4, the solution to which is developed through Sections 8.5 and 8.6. Results from this optimised packing are presented and discussed in Sections 8.7 and 8.8.

8.1 DOT-PRODUCT PARTIAL PRODUCT PACKING

The dot-product can be decomposed into a sum of significant (nonzero) partial products and packed to reduce the effective number of multiplications. The following example will be used to illustrate dot-product partial product packing (PPP).

The dot-product is to be calculated for two vectors $\mathbf{A}$ and $\mathbf{B}$ of length $L = 5$. Consider the example coefficients of $\mathbf{A}$ shown in Figure 8.1, where the five coefficients have word lengths of $N = 8$. The location of the zero-symbols is the important consideration at this point, so the symbol alphabet is left undefined. The significant symbols are denoted by $a_{i\langle j\rangle}$ and zero-symbols by 0.

Expanding the dot-product expression of (8.2) into a tabular form gives Figure 8.2(a). Each zero-symbol generates a zero partial product term in (8.2) that can be ignored in the accumulation, hence this reduces the dot-product to its significant components as

$\mathbf{a}_i$	$\mathbf{a}_0$	$\mathbf{a}_1$	$\mathbf{a}_2$	$\mathbf{a}_3$	$\mathbf{a}_4$
$a_{i\langle 0\rangle}$	$a_{0,\langle 0\rangle}$	0	$a_{2,\langle 0\rangle}$	0	$a_{4,\langle 0\rangle}$
$a_{i\langle 1\rangle}$	0	$a_{1,\langle 1\rangle}$	0	$a_{3,\langle 1\rangle}$	0
$a_{i\langle 2\rangle}$	$a_{0,\langle 2\rangle}$	$a_{1,\langle 2\rangle}$	0	$a_{3,\langle 2\rangle}$	$a_{4,\langle 2\rangle}$
$a_{i\langle 3\rangle}$	0	0	$a_{2,\langle 3\rangle}$	$a_{3,\langle 3\rangle}$	0
$a_{i\langle 4\rangle}$	$a_{0,\langle 4\rangle}$	0	$a_{2,\langle 4\rangle}$	$a_{3,\langle 4\rangle}$	0
$a_{i\langle 5\rangle}$	$a_{0,\langle 5\rangle}$	$a_{1,\langle 5\rangle}$	0	0	$a_{4,\langle 5\rangle}$
$a_{i\langle 6\rangle}$	0	$a_{1,\langle 6\rangle}$	0	$a_{3,\langle 6\rangle}$	0
$a_{i\langle 7\rangle}$	0	0	$a_{2,\langle 7\rangle}$	0	0

Figure 8.1: Example digits of the coefficient vector $\mathbf{A}$ where $a_{i\langle j\rangle}$ are significant nonzero digits.

in Figure 8.2(b).

Consider the traditional parallel array implementation for an 8-symbol multiplier from Section 7.3.2. All eight partial products are generated simultaneously and placed in an array with fixed shifts. The partial products in the array are summed in parallel. This parallel type of multiplier will implement one column of Figure 8.2(b) in one clock cycle. This implementation does not ignore the zero partial products since the addition tree's branch to that position already exists and ignoring it does not reduce the time or the resources used.

The proposed solution of PPP uses the typical parallel multiplier structure supplemented with a mechanism to relocate partial products along rows as shown by a $\Longleftarrow$ in Figure 8.3(a). This packs the partial products with the same fixed shift together, resulting in the dot-product table of Figure 8.3(b). To calculate the dot-product one column at a time now requires independent partial product generation, where previously the partial product units (PPUs) shared a common multiplicand. Accordingly, the resources required for a PPU will increase in the tradeoff for better performance. The out-of-order addition of partial products that results is still valid, since addition of integers or fixed-point reals is associative.

The multiplier hardware is augmented so that it still implements one column of the table per clock cycle. In Figure 8.3(b), the number of cycles required has been reduced from five to four by relocating partial products left and hence interleaving the multiplications. Notice that the third row, with ordinate $\langle 2\rangle$, has four partial products and is the longest with the only partial product in the fourth column. The rows directly above and below have only two partial products. A method to relocate one of the partial products from the long row to one of the short rows would reduce the number

		$\mathbf{a}_0 \times \mathbf{b}_0$	+	$\mathbf{a}_1 \times \mathbf{b}_1$	+	$\mathbf{a}_2 \times \mathbf{b}_2$	+	$\mathbf{a}_3 \times \mathbf{b}_3$	+	$\mathbf{a}_4 \times \mathbf{b}_4$
		$\text{pp}(a_{0\langle 0\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 0\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{2\langle 0\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 0\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 0\rangle}, \mathbf{b}_4)$
	+	$\text{pp}(a_{0\langle 1\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 1\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{2\langle 1\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 1\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 1\rangle}, \mathbf{b}_4)$
	+	$\text{pp}(a_{0\langle 2\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 2\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{2\langle 2\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 2\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 2\rangle}, \mathbf{b}_4)$
$\mathbf{A}\bullet\mathbf{B} =$	+	$\text{pp}(a_{0\langle 3\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 3\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{2\langle 3\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 3\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 3\rangle}, \mathbf{b}_4)$
	+	$\text{pp}(a_{0\langle 4\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 4\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{2\langle 4\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 4\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 4\rangle}, \mathbf{b}_4)$
	+	$\text{pp}(a_{0\langle 5\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 5\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{2\langle 5\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 5\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 5\rangle}, \mathbf{b}_4)$
	+	$\text{pp}(a_{0\langle 6\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 6\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{2\langle 6\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 6\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 6\rangle}, \mathbf{b}_4)$
	+	$\text{pp}(a_{0\langle 7\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 7\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{2\langle 7\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 7\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 7\rangle}, \mathbf{b}_4)$

(a) dot-product operation expanded in tabular form where a column is a multiplication expanded into partial products.

		$\mathbf{a}_0 \times \mathbf{b}_0$	+	$\mathbf{a}_1 \times \mathbf{b}_1$	+	$\mathbf{a}_2 \times \mathbf{b}_2$	+	$\mathbf{a}_3 \times \mathbf{b}_3$	+	$\mathbf{a}_4 \times \mathbf{b}_4$
		$\text{pp}(a_{0\langle 0\rangle}, \mathbf{b}_0)$			+	$\text{pp}(a_{2\langle 0\rangle}, \mathbf{b}_2)$			+	$\text{pp}(a_{4\langle 0\rangle}, \mathbf{b}_4)$
			+	$\text{pp}(a_{1\langle 1\rangle}, \mathbf{b}_1)$			+	$\text{pp}(a_{3\langle 1\rangle}, \mathbf{b}_3)$		
	+	$\text{pp}(a_{0\langle 2\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 2\rangle}, \mathbf{b}_1)$			+	$\text{pp}(a_{3\langle 2\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 2\rangle}, \mathbf{b}_4)$
$\mathbf{A}\bullet\mathbf{B} =$					+	$\text{pp}(a_{2\langle 3\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 3\rangle}, \mathbf{b}_3)$		
		$\text{pp}(a_{0\langle 4\rangle}, \mathbf{b}_0)$	+		+	$\text{pp}(a_{2\langle 4\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 4\rangle}, \mathbf{b}_3)$		
	+	$\text{pp}(a_{0\langle 5\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 5\rangle}, \mathbf{b}_1)$					+	$\text{pp}(a_{4\langle 5\rangle}, \mathbf{b}_4)$
			+	$\text{pp}(a_{1\langle 6\rangle}, \mathbf{b}_1)$			+	$\text{pp}(a_{3\langle 6\rangle}, \mathbf{b}_3)$		
					+	$\text{pp}(a_{2\langle 7\rangle}, \mathbf{b}_2)$				

(b) dot-product operation after removing the zero partial products generated by zero digits in (a)

Figure 8.2: Example of partial product packing to reduce the calculations performed in a dot-product operation with known coefficients in $\mathbf{A}$.

	$\mathbf{a}_0 \times \mathbf{b}_0$	+	$\mathbf{a}_1 \times \mathbf{b}_1$	+	$\mathbf{a}_2 \times \mathbf{b}_2$	+	$\mathbf{a}_3 \times \mathbf{b}_3$	+	$\mathbf{a}_4 \times \mathbf{b}_4$
	$\text{pp}(a_{0\langle 0\rangle}, \mathbf{b}_0)$	+	$\Longleftarrow$		$\text{pp}(a_{2\langle 0\rangle}, \mathbf{b}_2)$	+	$\Longleftarrow$		$\text{pp}(a_{4\langle 0\rangle}, \mathbf{b}_4)$
+	$\Longleftarrow$		$\text{pp}(a_{1\langle 1\rangle}, \mathbf{b}_1)$	+	$\Longleftarrow$		$\text{pp}(a_{3\langle 1\rangle}, \mathbf{b}_3)$		
+	$\text{pp}(a_{0\langle 2\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 2\rangle}, \mathbf{b}_1)$	+	$\Longleftarrow$		$\text{pp}(a_{3\langle 2\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 2\rangle}, \mathbf{b}_4)$
$\mathbf{A}\bullet\mathbf{B} =$ +	$\Longleftarrow$		$\Longleftarrow$		$\text{pp}(a_{2\langle 3\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 3\rangle}, \mathbf{b}_3)$		
+	$\text{pp}(a_{0\langle 4\rangle}, \mathbf{b}_0)$	+	$\Longleftarrow$		$\text{pp}(a_{2\langle 4\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 4\rangle}, \mathbf{b}_3)$		
+	$\text{pp}(a_{0\langle 5\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 5\rangle}, \mathbf{b}_1)$	+	$\Longleftarrow$		$\Longleftarrow$		$\text{pp}(a_{4\langle 5\rangle}, \mathbf{b}_4)$
+	$\Longleftarrow$		$\text{pp}(a_{1\langle 6\rangle}, \mathbf{b}_1)$	+	$\Longleftarrow$		$\text{pp}(a_{3\langle 6\rangle}, \mathbf{b}_3)$		
+	$\Longleftarrow$		$\Longleftarrow$		$\text{pp}(a_{2\langle 7\rangle}, \mathbf{b}_2)$				

(a) Dot-product with the zero partial products removed and $\Longleftarrow$ showing the relocation of partial products with a similar hard-wired shift.

	$\mathbf{a}_{0/1/2} \times \mathbf{b}_{0/1/2}$	+	$\mathbf{a}_{1/2/3} \times \mathbf{b}_{1/2/3}$	+	$\mathbf{a}_{3/4} \times \mathbf{b}_{3/4}$	+	$\mathbf{a}_4 \times \mathbf{b}_4$
	$\text{pp}(a_{0\langle 0\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{2\langle 0\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{4\langle 0\rangle}, \mathbf{b}_4)$		
+	$\text{pp}(a_{1\langle 1\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{3\langle 1\rangle}, \mathbf{b}_3)$				
+	$\text{pp}(a_{0\langle 2\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 2\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{3\langle 2\rangle}, \mathbf{b}_3)$	+	$\text{pp}(a_{4\langle 2\rangle}, \mathbf{b}_4)$
$\mathbf{A}\bullet\mathbf{B} =$ +	$\text{pp}(a_{2\langle 3\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 3\rangle}, \mathbf{b}_3)$				
+	$\text{pp}(a_{0\langle 4\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{2\langle 4\rangle}, \mathbf{b}_2)$	+	$\text{pp}(a_{3\langle 4\rangle}, \mathbf{b}_3)$		
+	$\text{pp}(a_{0\langle 5\rangle}, \mathbf{b}_0)$	+	$\text{pp}(a_{1\langle 5\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{4\langle 5\rangle}, \mathbf{b}_4)$		
+	$\text{pp}(a_{1\langle 6\rangle}, \mathbf{b}_1)$	+	$\text{pp}(a_{3\langle 6\rangle}, \mathbf{b}_3)$				
+	$\text{pp}(a_{2\langle 7\rangle}, \mathbf{b}_2)$						

(b) A compressed dot-product operation that requires fewer but more complicated merged multiplications.

Figure 8.3: Example of partial product packing to reduce the calculations performed in a dot-product operation with known coefficients in $\mathbf{A}$.

of columns to three. Only three clock cycles would then be required, reducing the dot-product latency.

A novel method for achieving this relocation is through the careful selection of redundant number representations for the coefficients in **A**. The development of this original method and its associated combinatorial optimisation algorithm is presented in Sections 8.2 through 8.8. The hardware design that implements the PPP is presented in Chapter 9.

8.2 BENEFITS OF USING REDUNDANT REPRESENTATIONS

A common application of redundant number systems is their use in building carry-propagation-free adders. Eliminating carry propagation means that two values can be added in constant time, irrespective of their word length. In a multiplier this can be utilised to quickly sum partial products in an addition tree and accumulate the result. Although not explicitly thought of as using redundant number representations, the addition trees of Wallace [1964] and Dadda [1977] used in binary multipliers have redundant intermediate results and convert to binary, the non-redundant form, at their root. In the hardware implementation presented in Section 9.3.3, constant time addition is used for both the partial product addition in the multiplier and for the accumulation. If required, the relatively slow conversion to binary, that requires carry-propagation, only needs to be performed once at the completion of the dot-product. It can be pipelined with the dot-product, running at slower clock rate than the rest of the design as shown in Section 9.4.

A more explicit use of redundant number systems is in multipliers to reduce the number of nonzero partial products. This is achieved by recoding the coefficients of **A** to minimise their Hamming weight, for example using canonical signed digit (CSD) [Soderstrand et al., 2000]. Booth recoding [Booth, 1951, Sam and Gupta, 1990] is an initial application of this, reducing the number of partial products to half the word length. Reducing the Hamming weight of each coefficient in **A** provides a substantial reduction in the number of partial products, hence fewer columns after packing.

Even with the reduced Hamming weight representations, a PPP situation with uneven row lengths is still likely to occur. As illustrated in Figure 8.3(b), a single partial product is present in the fourth column and multiple slots are free in the third column. We note that the column count could be reduced further if this lone partial product could be 'moved' into one of the slots. Assuming a fixed shift architecture for the implementing hardware, the partial products must stay in their rows, so moving the partial product vertically is not an option.

8.3 METHODS TO IMPROVE PACKING

The immediately obvious solution to uneven row lengths is to split the undesirable partial product with shift j into the sum of P lower positioned partial products,

$$\mathrm{pp}(a_{i\langle j\rangle}, \mathbf{b}_i) = \sum_{p=0}^{P-1} (\mathbf{s}(p)\mathbf{b}_i)\, \beta^{h(p)}. \tag{8.5}$$

That is to say, $a_{i\langle j\rangle}$ has been replaced by P new symbols,

$$a_{i\langle j\rangle} = \mathbf{s}(0)_{i,\langle h(0)\rangle} + \mathbf{s}(1)_{i,\langle h(1)\rangle} + \cdots + \mathbf{s}(P)_{i,\langle h(P)\rangle}. \tag{8.6}$$

To remain consistent with the number system used, the replacement symbols ($\mathbf{s}(p)$) must be chosen from $\mathbb{S}$ and have fixed shifts of $0 \leq h(p) < N$. To achieve the desired result of reducing the number of partial products in the $\langle j\rangle$th row by one, the replacing partial products need to have a shift $h(p) \neq j$. A simple example is replacing the partial product with β copies of itself each right shifted by one ($\times\beta^{-1}$),

$$\mathrm{pp}(a_{i\langle j\rangle}, \mathbf{b}_i) = \sum^{\beta} \mathrm{pp}(a_{i\langle j\rangle}, \mathbf{b}_i)\beta^{-1}. \tag{8.7}$$

If $\beta = 2$ this is equivalent to the replacement of $a_{i\langle j\rangle}$ by,

$$a_{i\langle j\rangle} = \frac{\mathbf{a}_{i,\langle j\rangle}}{2} + \frac{\mathbf{a}_{i,\langle j\rangle}}{2} \tag{8.8}$$

This solution creates P new partial products that necessarily increase the lengths of other rows. In the example $P = \beta$. While some extra partial products can be absorbed, the number is tightly limited. If $\mathbf{R} = (R_0, R_1, \cdots, R_{N-2}, R_{N-1})$ is a vector of the row lengths and $R_{\max} = \max(\mathbf{R})$ is the 'row ceiling', then the number of unused partial product slots is equal to $(N)R_{\max} - \sum_{i=0}^{N-1} R_i$. Each time this method is successfully applied, $R_{\max}$ is reduced by one, thus reducing the number of empty slots by $N-1$ and P slots are consumed. Hence, $N-1+P$ empty slots are quickly eliminated. Therefore, this method is only useful in a few situations and obviously cannot be employed successfully en masse for a larger dot-product as too many new partial products are created.

A novel and better solution relies on the redundancy in the coefficients' representations. Initially, a redundant number system was employed to reduce the coefficients' Hamming weight and hence reduce the number of partial products. Consider a simple example with two coefficients that have the same value (coefficients are often duplicated in real valued FIR filters). If the values are generated by a Hamming weight minimising linear recoding [Phillips and Burgess, 2004], then both coefficients will have the same

$343_{10} =$	1	0	1	0	0	3	0	0	$\overline{1}_2$
$343_{10} =$	1	0	1	0	0	3	0	0	$\overline{1}_2$
place	•		•			•			•
usage	•		•			•			•

(a) Simple linear minimum Hamming weight recoding.

$343_{10} =$	1	0	1	0	0	3	0	0	$\overline{1}_2$
$343_{10} =$	0	3	0	$\overline{1}$	0	$\overline{1}$	0	0	$\overline{1}_2$
place						•			•
usage	•	•	•	•		•			•

(b) Representations from the minimum Hamming weight set.

$343_{10} =$	1	0	0	3	$\overline{1}$	0	1	0	3_2
$343_{10} =$	0	3	$\overline{1}$	0	0	3	0	0	$\overline{1}_2$
place									•
usage	•	•	•	•	•	•	•		•

(c) Representations from the zero dominant set.

Figure 8.4: Examples of packing of two equal valued (343_{10}) coefficients using its redundant representations selected from different sets (reproduction of Figure 4.1).

pseudo-redundant representation. No packing is achieved because the nonzero symbols are in the same place, as shown in Figure 8.4(a). This is because the recoding is a simple one-to-one mapping. If the coefficient's representations can be chosen from the set of all minimum Hamming weight representations, then some packing may be achieved, as shown in Figure 8.4(b). While a coefficient may have several minimum Hamming weight representations, it has many representations with slightly higher weight. For example, 23_{10} whose representations in $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$,$\beta = 2$ include $0300\overline{1}_2$, 10031_2, and 10103_2. Most importantly, the positions of the zero-symbols vary between these representations. With an unrestricted representation choice, a better packing can be achieved, even with an increase in the total number of nonzero partial products as shown in Figure 8.4(c).

Appropriately selecting representations for the coefficients of $\mathbf{A}$ would allow zero-symbols to be concentrated into a particular row, thus reducing the length of that row. Judicious application of this technique would allow all rows to be balanced in length. In the example of Figure 8.3(b), finding a representation for $\mathbf{a}_4$ that has zero symbols in rows $\langle 2 \rangle$ and $\langle 4 \rangle$, the rows with three partial products from *other* coefficients, will give a row length ceiling of three instead of four. The fourth column of Figure 8.3(b) is

eliminated and some of rows $\langle 1 \rangle$, $\langle 3 \rangle$, $\langle 6 \rangle$, and $\langle 7 \rangle$ will become more full. The number of cycles to calculate the dot-product is now reduced to 3 giving a further performance improvement.

If such a representation for $\mathbf{a}_4$ does not exist, the same effect may be achieved by finding an alternate representation for another coefficient such that it has a zero symbol in the long row. Again, using the example of Figure 8.3(b), one of $\mathbf{a}_0$, $\mathbf{a}_1$, or $\mathbf{a}_3$ may have an alternate representation with a zero symbol in row $\langle 2 \rangle$, thus opening a slot for $\text{pp}(a_{4\langle 2 \rangle}, \mathbf{b}_4)$ to be packed into. Furthermore, a co-ordination between the representation chosen for each coefficient may be required to achieve this. Indeed, this co-ordination is critical in finding a good packing solution.

8.4 PARTIAL PRODUCT PACKING PROBLEM

The solution presented above poses a large combinatorial optimisation problem:

> *Given a set of coefficient values and a redundant number system, which selection of representations, one for each coefficient, minimises the row ceiling in the packed partial product?*

The size of the problem is characterised by the number of representations for each coefficient. On average an N symbol coefficient has $O(|\mathbb{S}|^N/\beta^N)$ representations for each value, where $|\mathbb{S}|^N$ is the number of permutations for N symbols chosen from the symbol alphabet $\mathbb{S}$, and β^N is the approximate size of the representable integer range with N symbols. Therefore, a PPP problem with L coefficients has a search space of $O((|\mathbb{S}|/\beta)^{NL})$ combinations of representations. Even for a small dot-product of 21 coefficients from a small redundant number system with $|\mathbb{S}| = 3$, $\beta = 2$, and $N = 16$, the number of combinations is $(3/2)^{16\times 21} \approx 657^{21} \approx 10^{59}$. Finding a deterministic solution by trying all combinations is infeasible.

Finding the best combination of redundant coefficient representations so the maximum row length is minimised is suspected to be a non-deterministic polynomial-time (NP) problem. An NP problem is one where a proposed solution can be *verified* in polynomial-time, but the solution cannot be *found* with a deterministic method in polynomial-time [Cook et al., 1998]. Polynomial-time refers to the time to find the solution as a finite polynomial function of n, the size of the problem, in this case the number of coefficients ($n \equiv L$).

The PPP optimisation problem can be stated as:

> *Find a set of representations, for the given coefficients, such that the maximum row length after partial product packing is less than l,*

where l is an arbitrarily small target and $0 < l \leq L$. This target can be verified for a proposed solution in polynomial-time, i.e., perform the partial product packing and check $R_{\max} \leq l$.

8.5 ZERO-DOMINANT SET

Most of a coefficient's representations will be useless in the partial packing problem, or their usefulness is replicated and improved by another representation. Minimum Hamming weight is too strict a condition to provide optimal packing, see Figure 8.4. The set of representations to consider are those that provide unique zero-symbol positioning while minimising the Hamming weight. These are the zero-dominant representations described in Section 4.2. The algorithm to generate a coefficient's zero-dominant representation set is presented in Section 4.4 and called in Listing 8.1 as BuildZDS(). Given that at least one representation satisfies a particular constraint on the zero-symbol positions, there will be a zero-dominant representation that also maximises the number of zero-symbols. The zero-dominant set for value v, denoted $\mathbb{D}_v$, has a representation for each possible positioning of zero-symbols. Some representations will cover many constraints, in a Pareto-optimal sense. Therefore, $\mathbb{D}_v$ contains as few representations as possible to cover all practical constraints on zero-symbol positions.

The size of the zero-dominant set is dependent on the value being represented and the redundant number system employed. In general, smaller values have smaller zero-dominant sets, since the zero-dominance rules promote those representations that zero extend as soon as possible, i.e, have zero symbols in the most significant positions. This means $|\mathbb{D}_v|$ is largely independent of the word length N.

8.6 PARTIAL PRODUCT PACKING OPTIMISATION ALGORITHM

Each coefficient in a vector $\mathbf{A}$ requires a representation. The algorithm presented in Listing 8.1 seeks a single representation for each coefficient such that the row length ceiling in minimised. A collision is said to occur between the representations of two different coefficients when they both have a nonzero symbol in the same position. The

algorithm attempts to minimise the number of collisions occurring in each row. The algorithm listed in Listing 8.1 can find a near optimal solution to the PPP problem in polynomial-time.

The collision distance, c, calculated at line 21 is the L-norm of the row lengths corresponding to nonzero symbols in representation being tested (rep). Increasing the order of the norm $(L^1, L^2, L^3, \ldots)$ reduces the preference for minimum Hamming weight representations. This exposes representations with slightly greater Hamming weight that have an alternate and possibly better placement of zero symbols.

The algorithm checks each coefficient in turn to see if another representation has fewer collisions than the current representation for a particular a_i; this takes a constant time for a given word size. Therefore, the algorithm has computational complexity linear in the total number of representations, $O(\sum_{i=0}^{|\mathbf{A}|-1} |\mathbb{D}_i|)$.

8.7 RESULTS AND DISCUSSION

To test the performance of the PPP optimisation algorithm, a number of Monte-Carlo trials were run where the vector $\mathbf{A}$ was varied in both value and length. The length was varied in between 20 and 200 inclusive steps of 5 to give odd and even length filters of reasonable length. The coefficient values had a word length of 16, with range $-2^{15} \leq x \leq 2^{15}$. The coefficients were regenerated for each test from either of two sources:

1. Uniform random distribution, or
2. Low pass FIR filter designed using the Remez exchange algorithm with a randomised cut-off frequency.

The details of these coefficient sources and their resulting packing are presented in Sections 8.7.3 and 8.7.4 respectively.

The redundant number system was chosen to have a symbol alphabet $\mathbb{S}$ with a maximum cardinality of four and a radix of 2. This provides sufficient redundancy while limiting the growth of resources required in a practical hardware implementation. This will be explained further in Chapter 10. The symbol alphabet is chosen to minimise the average minimum Hamming weight. The number system used is $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$ with $\beta = 2$. It has an average minimum Hamming weight of $2N/7 + 8/49$ as $N \to \infty$, as will be shown in Section 10.4. This means a 16 symbol number will have on average 4.7 nonzero symbols in its minimum Hamming weight representation.

Listing 8.1: An algorithm to pack the partial products in the dot-product with the vector **A** of constant coefficients. The representations are generated for the redundant number system with alphabet $\mathbb{S}$, and radix β.

```
1: procedure PartialProductPack(A, 𝕊, β, N)
2:     D ← {}                          ▷ List of representation sets, one for each a_i in A.
3:     d ← {}                          ▷ List of packed representations, one for each a_i in A.
4:     R ← {0 for n in 0 to N − 1}      ▷ Initialise row length vector to zeros.
5:     for each a_i ∈ A do                                          ▷ Populate D.
6:         D_i ←BuildZDS(a_i, 𝕊, β)      ▷ Find the zero-dominant set for a_i.
7:         d_i ← random representation from D_i
8:         R ← updateR(R, d_i, 1)                                   ▷ Count row lengths.
9:     end for
10:    i ← 0
11:    last ← None
12:    while i ≠ last do    ▷ Continue until no coefficient changes its representation.
13:        for each a_i ∈ A do
14:            R ← updateR(R, d_i, −1)        ▷ Subtract d_i's contribution from R.
15:            c_min ← N² × |A|                ▷ Initialise c_min to be large.
16:            best ← None
17:            for each rep ∈ D_i do
18:                c ← 0                              ▷ Initialise collision distance.
19:                for n in 0 to N − 1 do          ▷ Calculate the collision distance.
20:                    if rep_⟨n⟩ ≠ 0 then
21:                        c ← c + R_n²                                  ▷ L² Norm.
22:                    end if
23:                end for
24:                if c < c_min then          ▷ Is this representation the best so far?
25:                    c_min ← c
26:                    best ← rep
27:                end if
28:            end for
29:            if best is not d_i then          ▷ Was a better representation found?
30:                d_i ← best                ▷ Update the coefficients representation d_i.
31:                last ← i      ▷ Record the last coefficient to change representation.
32:            end if
33:            R ← updateR(R, best, 1)
34:        end for
35:        i ← (i + 1) mod |A|                                      ▷ Next coefficient.
36:    end while
37:    return d
38: end procedure
39:
40: procedure updateR(R, rep, score)
41:    for n in 0 to |rep| − 1 do
42:        if d_{i,⟨n⟩} ≠ 0 then
43:            R_n ← R_n+ score
44:        end if
45:    end for
46:    return R
47: end procedure
```

8.7.1 Irreducible first row

When using a number system with integer symbols, the least significant row (with positional weight $\beta^0 = 1$) is irreducible. All coefficient values with $v \bmod \beta \neq 0$ require a symbol in this position. Therefore, the number of coefficients in $\mathbf{A}$ with $v \bmod \beta \neq 0$ dictates the number of partial products in row $\langle 0 \rangle$. Assuming uniform probability for the random variable $V = a_i \bmod \beta$, the expected length of the least significant row will be $\mathrm{E}(R_0) = L(\beta - 1)/\beta$. For $\beta = 2$, the least significant row will have an expected length $L/2$; the expected number of odd coefficients.

As will be shown, for $\beta = 2$, the optimal PPP result usually achieves row lengths better than $L/2$ at all other rows. To stop the least significant row dominating the row length ceiling, the partial products in the least significant row are shared among α_0 rows of the same shift. The hardware implements α_0 copies of the PPU with no shift. This approach is assumed for the remainder of this chapter. The least significant row's length is reported as $R'_0 = \lceil R_0/\alpha_0 \rceil$, where usually $\alpha_0 = 2$.

8.7.2 Shadow optimum

Determining the optimum packing is a hard combinatorial problem, therefore, the minimum row length ceiling R_{opt} is unknown. A lower bound on R_{opt}, is used to assess the optimality of the solutions provided by the PPP optimisation algorithm. This is designated the shadow optimum, R_s.

The shadow optimum is estimated from two parameters. The first parameter R_s^1 considers all rows individually and asks; *what is the minimum number of partial products for this row?* That is, for each row, count the coefficients that must have a partial product in this row, because none of the coefficient's zero-dominant representations have a zero at the position corresponding to that row. The parameter R_s^1 is the maximum of these counts. The optimum solution can never have a lower ceiling than this.

The second parameter R_s^2 is the row length ceiling when each coefficient is represented by a minimum Hamming weight representation and each row is used equally. If the dot-product is initially populated with minimum Hamming weight representations then,

$$R_s^2 = \left\lceil \frac{\sum_{n=0}^{N-1} R_n}{N + \alpha_0 - 1} \right\rceil, \tag{8.9}$$

where R_n is the length of row $\langle n \rangle$ and α_0 is the number of physically implemented rows that row $\langle 0 \rangle$ is split into as defined in Section 8.7.1.

Equation (8.9) gives a close estimate of the global optimum when the probability of partial product placement is uniform over all rows, such as with uniform random coefficients. However, for a non-uniform distribution, particularly where smaller values have higher probabilities, R_s^2 gives a poor estimate since the rows are not equally used. This is revisited in the section 8.7.4 where non-uniformly distributed coefficients from a FIR impulse responses are considered.

The shadow optimum R_s is thus the maximum of the two lower bounds,

$$R_s = \max(R_s^1, R_s^2). \tag{8.10}$$

Note that R_s^2 is the ultimate goal of the algorithm, however, R_s^1 may make that impossible.

8.7.3 Partial product packing experiment with random coefficients

To test how much PPP reduces the effective number of multiplications and the further improvement provided by the optimisation algorithm, a series of experimental trials were run as described in Section 8.7. For the first series of trials, the integer coefficients in vector $\mathbf{A}$ are randomly generated from the discrete uniform distribution $U(-2^{15}, 2^{15})$ and encoded using the redundant number system with $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$ and $\beta = 2$. The dot-product was initially populated using the coefficients' representations randomly chosen from those with minimum Hamming weight. Packing the dot-product at this initial stage gave a distribution of the maximum row lengths shown in Figure 8.5 by the white histograms.

The dot-product coefficients were then optimised using the PPP algorithm. This process reduces both the mean and spread of the row length ceiling ($R_{\max}$) distributions, as shown in Figure 8.5. As can be seen in Figure 8.6, the mean packing, $\overline{R_{\max}}$, increases linearly with the vector length L. The slope is approximately the average minimum Hamming weight per position, $E(H_{\min})/N \approx 2/7$ [Phillips and Burgess, 2004]. The spread also increases slowly with L, but plateaus at higher vector lengths.

The shadow optimum was calculated for each dot-product to assess the optimality of the achieved row ceilings. The white histograms in Figure 8.7 show the distribution of differences between the shadow optimum and the initial row length ceiling, before optimisation. It suggests that it is unlikely that an optimum solution will be encountered at this initial stage. Also, as the number of coefficients increases so does the mean distance from the optimum packing. The black histograms show the distribution of differences between the shadow optimum and the post-optimisation ceiling. Comparing

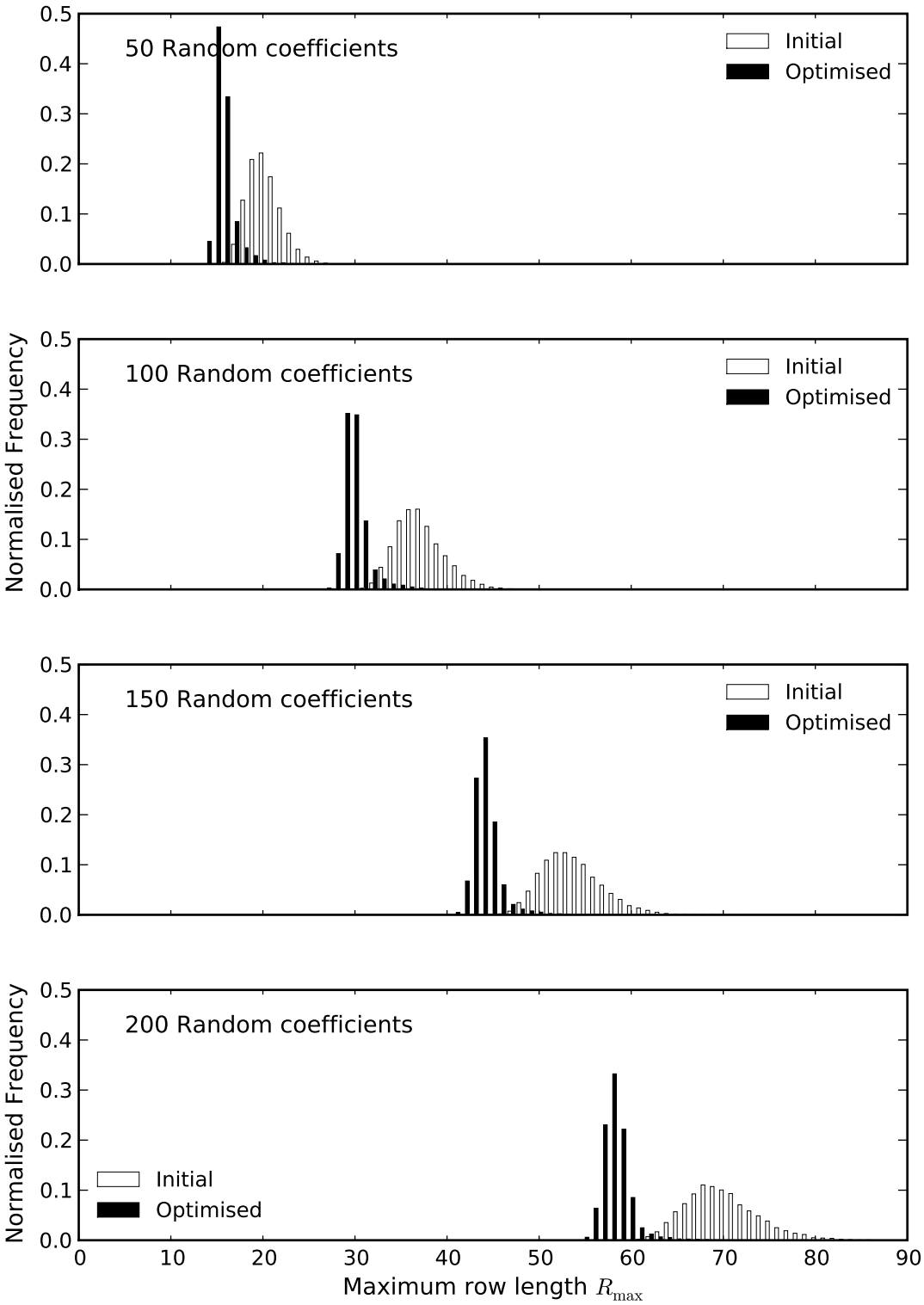


Figure 8.5: A series of histogram pairs showing the row length ceiling of partial product packed vectors before and after optimisation. The coefficients are uniformly distributed random numbers and the vector lengths vary from 50 to 200. Normalised results from 10,000 trials.

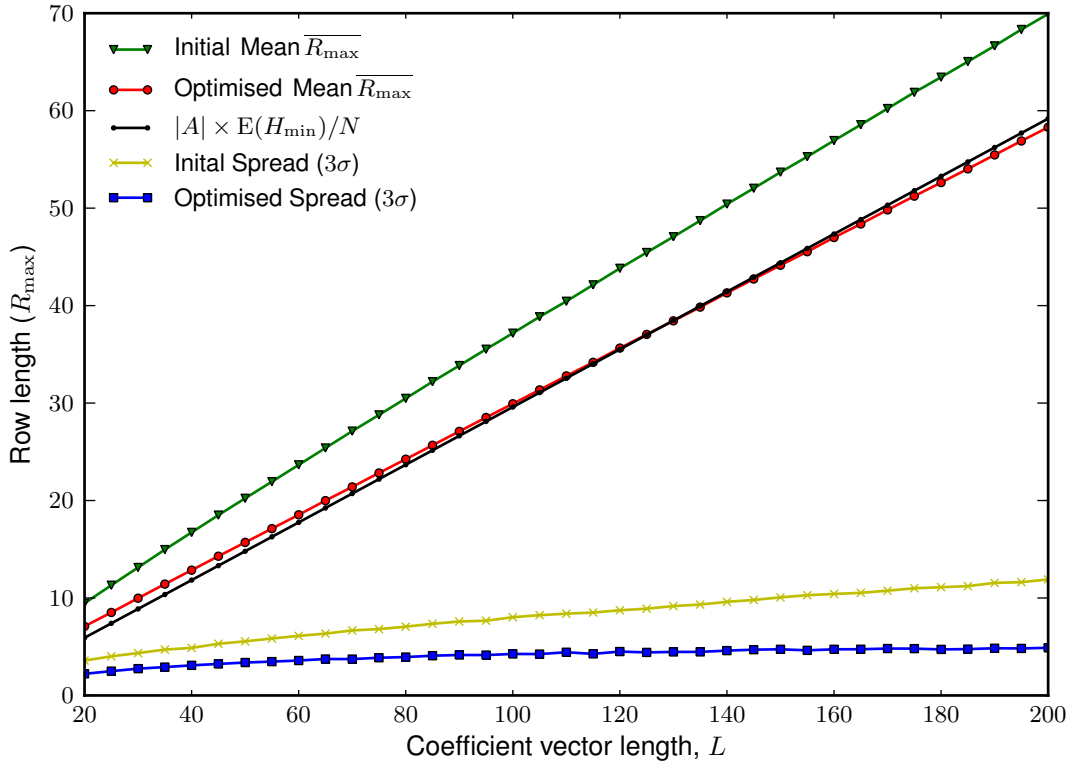


Figure 8.6: Mean and range of $R_{\max}$ for uniformly distributed coefficients. Results derived from 10,000 trials.

the histogram pairs shows that the algorithm of Listing 8.1 has significantly improved the optimality of the packing. The mean and spread of the distributions have been decreased and often the shadow optimum has been achieved, i.e., zero difference between the shadow optimum and the optimised packing. Remember that the shadow optimum R_s is only a *lower* bound estimate of the global optimum row ceiling R_{opt}. Therefore, when $R_{\text{opt}} > R_s$ then the PPP optimisation algorithm has performed better at finding a solution than indicated by the results of Figure 8.7 comparing R_{max} to R_s.

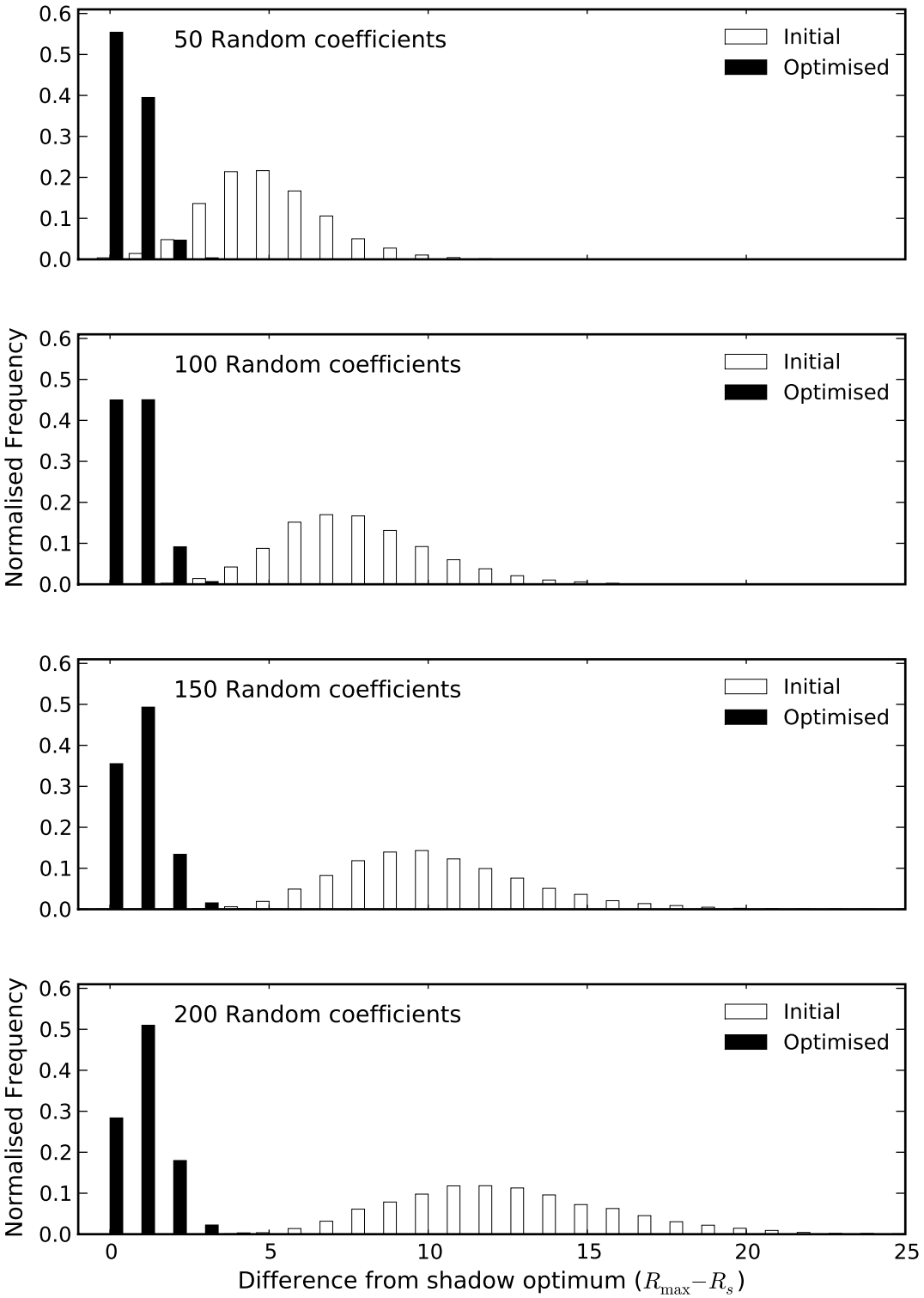


Figure 8.7: A series of histogram pairs showing the difference between the shadow optimum and the achieved row length ceiling of partial product packed vectors before and after optimisation. The coefficients are uniformly distributed random numbers and the vector lengths vary from 50 to 200. Normalised results from 10,000 trials.

8.7.4 Partial product packing experiment with FIR filter coefficients

A typical application of a dot-product in DSP is the filtering of a signal using a linear-phase finite impulse response (FIR) filter. This application provides an additional challenge to the PPP algorithm. The coefficients for a FIR filter, also known as taps, are usually duplicated since the impulse response is symmetric. Improved packing may be achieved by using different redundant representations for the duplicate values, as demonstrated earlier in Figure 8.4. The optimisation algorithm should treat the identical coefficients separately, and if required, choose different representations for them. Other attempted optimisation algorithms suffered sub-optimal packing by treating the twin coefficients identically and giving them the same representation.

For the second series of trials, the coefficients in vector $\mathbf{A}$ were FIR filter coefficients. These were designed using the Remez exchange [McClellan et al., 1973] algorithm in the Python SciPy module [Jones et al., 2001]. The filters were all low-pass with a randomly chosen passband edge between $0.1 \times \mathrm{f_s}$ and $0.4 \times \mathrm{f_s}$, where $\mathrm{f_s}$ is the sampling frequency. The passband ripple, δ_p, and stopband ripple, δ_s, were set at $-40\,\mathrm{dB}$. The number of taps, L, was varied between 25 and 200 as in the random coefficient series of trials. The transition bandwidth ΔF was adjusted according to L so that the impulse response did not fade to below the quantisation level before the desired filter length was achieved. The transition bandwidth was calculated from a rearrangement of an empirical formula for estimating the number of taps required [Herrmann et al., 1973, Ifeachor and Jervis, 2002]. After rearrangement this gives,

$$\Delta F = \frac{1 - L \pm \sqrt{(1-L)^2 + 4f(\delta_p, \delta_s) D_\infty(\delta_p, \delta_s)}}{2f(\delta_p, \delta_s)}, \tag{8.11}$$

where the positive solution is used.

Initially the coefficients' representations were chosen randomly from those with minimum Hamming weight. This gave the initial distribution of maximum row lengths shown by the white histograms in Figure 8.8. Like the random coefficient trials, the mean and spread or $R_{\max}$ are reduced by the PPP optimisation algorithm. The optimised row lengths are shown by the black histograms in Figure 8.8.

An interesting feature is present in the optimised histograms. Even row ceilings occur with greater frequency for vectors with even lengths. The distributions also have a larger spread compared to the random coefficient distributions. These two effects are attributed to the duplication of coefficients, where there is a lack of representation diversity in the zero-dominant sets. This is where the collision between twin coefficients cannot be avoided, like in Figure 8.4.

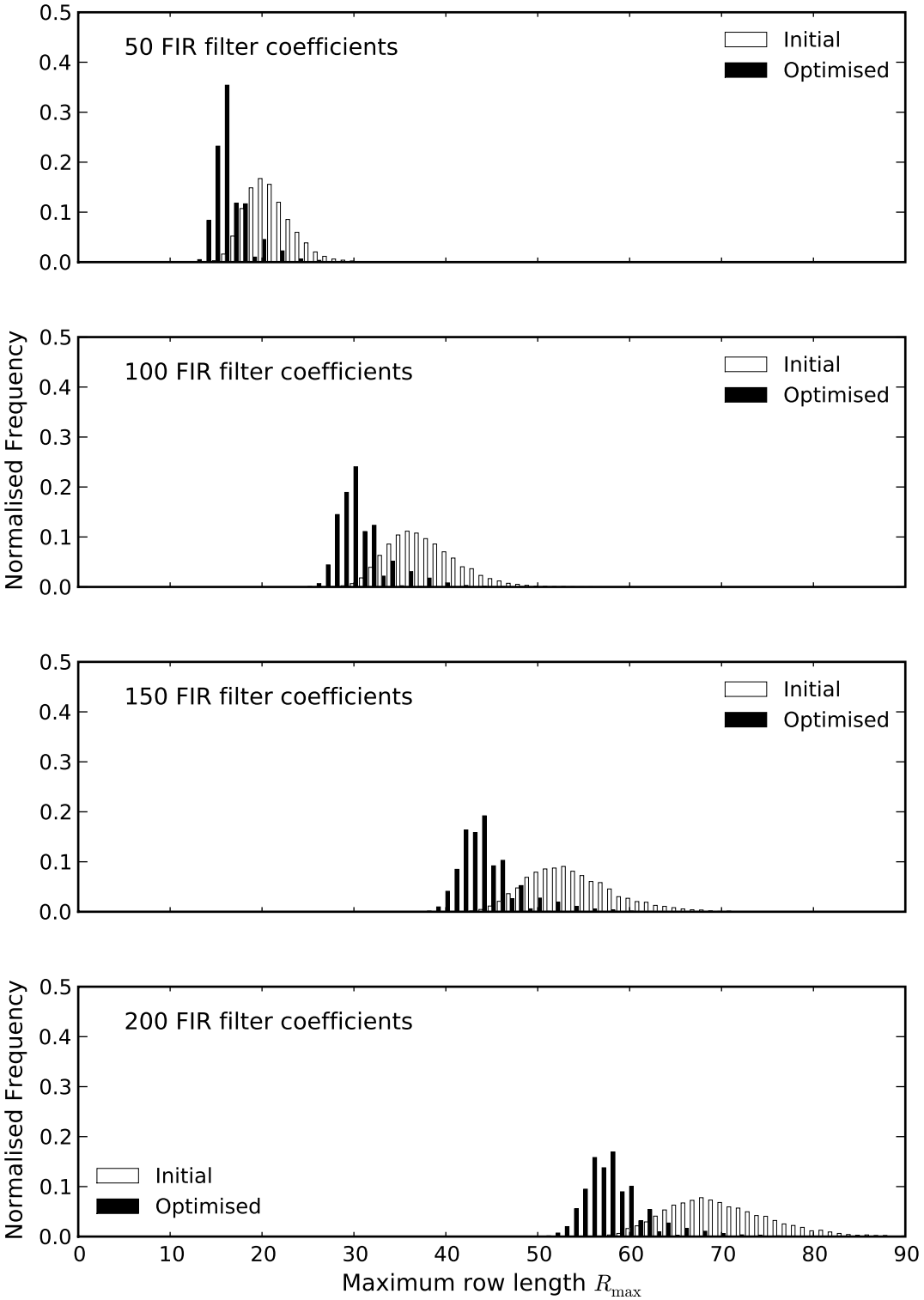


Figure 8.8: A series of histogram pairs showing the row length ceiling of partial product packed vectors before and after optimisation. The coefficients are from low-pass FIR filters with tap lengths of between 50 and 200. Normalised results from 10,000 trials.

The optimum maximum row length is difficult to estimate in the case of a FIR filter. The filter's sinc-like[1] impulse response consists of many more small values than large values, since the magnitude decays as $1/(\pi x)$. This means there are naturally fewer partial products in the more significant rows. This skewing makes R_s^2 a poor lower bound for R_{opt}, as it underestimates the length of the least significant rows and overestimates it for the more significant rows. To obtain a tighter bound consider only rows $\langle 1 \rangle$ to $\langle \lceil N - \log_2 \left(\frac{L}{2}\pi\right) \rceil \rangle$ inclusive. The upper row limit is calculated from the magnitude of the $1/(\pi x)$ envelope at the filter start and end taps where $x = \pm L/2$. All coefficients will have approximately equal probability of partial products in these least significant rows, thereby restoring the assumption of R_s^2. Equalising these least significant rows is now the target. Row $\langle 0 \rangle$ is excluded because the number of partial products is fixed as explained in section 8.7.1.

The shadow optimum was calculated for each vector using the above modification and compared with the initial and post-optimisation row ceilings. The distribution of these differences are plotted in Figure 8.9. The white histograms again show the distribution of differences between the shadow optimum and the initial row length ceilings before optimisation. Similarly the black histogram shows the distribution of differences between the shadow optimum and the post-optimisation ceiling. There is a high incidence of the packed lengths meeting the shadow optimum. This is where the first shadow parameter, R_s^1, is maximum in (8.10). This happens more often with FIR filter coefficients because the values are duplicated.

In comparison with the random coefficient trials, the Remez coefficient PPP did not achieve the same shadow difference performance. The FIR coefficient distributions have approximately double the spread. This may be due largely to the second shadow parameter R_s^2 being a looser lower bound than for the random coefficients.

8.7.5 Stochastic partial product packing algorithm

The algorithm developed and presented in Listing 8.1 takes a greedy path towards the optimum solution with a strict improvement criterion [Cook et al., 1998]. This gives a very quick convergence to a solution, but it is liable to become stuck in a local minimum of the optimisation landscape. To improve the search for the global minimum, the algorithm was augmented with ideas from the simulated annealing optimisation method [Kirkpatrick et al., 1983]. Firstly, the coefficient to be inspected is randomly selected with uniform probability. This is in contrast with Listing 8.1 where the coefficients are selected in a round robin. Secondly, when the collision distance, c, of a representation is calculated, it is converted into a probability of that representation being selected.

[1] a sinc function has the formula $\sin(\pi x)/(\pi x)$.

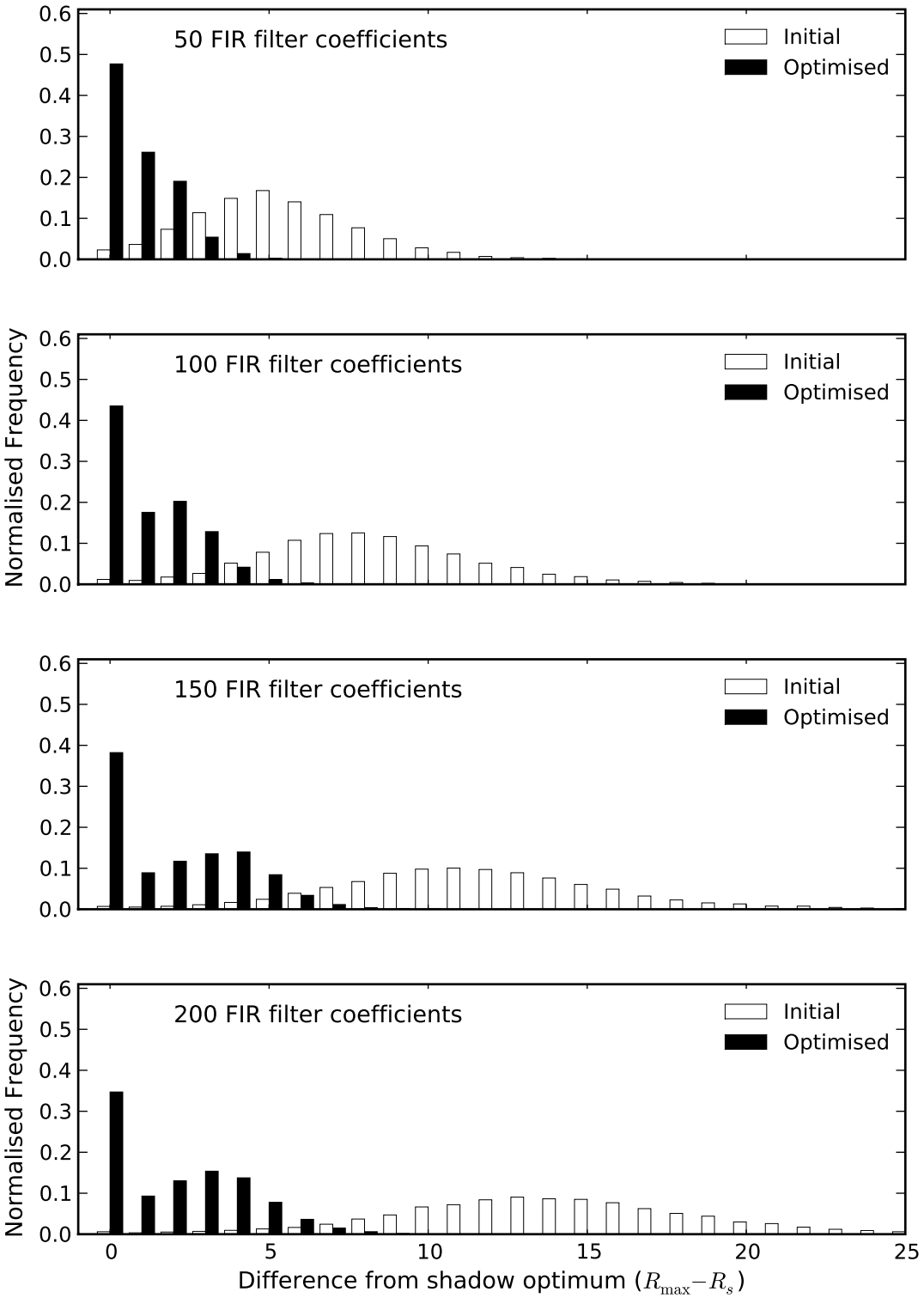


Figure 8.9: A series of histogram pairs showing the difference between the shadow optimum and the achieved row length ceiling of partial product packed vectors before and after optimisation. The coefficients are from low-pass FIR filters and the vector lengths of 50, 100 and, 200. Normalised results from 10,000 trials.

Table 8.1: The results of trials comparing the stochastic optimisation method to the greedy optimisation method. The table shows the percentage of trials where the stochastic method provided a better packing for the same random vector of coefficients. Packing was only improved by one column at the most.

L	% of trials where stochastic improved greedy
25	$224/1500 = 15\%$
51	$270/1500 = 18\%$
100	$354/1500 = 23.6\%$

Representations with a lower collision distance have a higher probability of selection. As the optimisation process progresses the selection probability is adjusted to further favour lower collision distances. This is synonymous to the annealing process.

It was found experimentally that the stochastic optimisation method reduced $R_{\max}$ by up to one column over the greedy method. Table 8.1 shows the results of comparative trials between the stochastic and greedy methods. The stochastic method required approximately 50 times more iterations, each with the increased computational cost of performing selection based on non-static probabilities. The greedy method reached the same result in the majority of trials and it may be preferred for its simplicity and speed.

8.8 SUMMARY

Partial product packing is a way of reducing the effective number of multiplications in a dot-product operation. By allowing partial products from multiple coefficients to be accumulated concurrently, and ignoring the zero partial products, the number of multiply accumulate cycles is reduced. However, this requires the multiplier structure to independently generate partial products from different multiplications concurrently. Redundant number systems with a low Hamming weight can be employed to increase the number of zero partial products, thereby reducing the effective number of multiplications.

A novel opportunity was identified to use the extra redundancy in the coefficients' representations to further reduce the effective number of multiplications. Through careful selection of a redundant representation for each dot-product coefficient, the efficiency of the PPP can be maximised. Attempting to achieve the fewest multiplier cycles per dot-product is a large combinatorial optimisation problem. A fast greedy heuristic algorithm was presented to make this selection in a near optimal way. This was shown to work by comparison with the shadow optimum, an estimated lower bound to the global optimum PPP. The greedy algorithm was also compared to an improved

stochastic version that uses simulated annealing. This improved the packing by only one column in 15-25% of cases, varying with the vector length.

Since the optimisation of the PPP is an involved and time consuming process, it is best suited to dot-products that have one constant vector of coefficients, such as in a FIR filter. The coefficients may be changed occasionally, such as in a block-adaptive FIR filter.

Chapter 9

IMPLEMENTATION OF A MULTIPLY ACCUMULATE UNIT

Digital signal processing (DSP) hardware commonly incorporates a multiply and accumulate (MAC) unit to make dot-product evaluation fast and efficient. One MAC operation is the product of two operands $\mathbf{a}_i$ and $\mathbf{b}_i$ added to a third, $\mathbf{c}_i$, the output of the previous MAC operation,

$$\mathbf{c}_{i+1} = \mathrm{MAC}(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i) = \mathbf{a}_i \times \mathbf{b}_i + \mathbf{c}_i. \tag{9.1}$$

The dot-product operation is the repeated application of the MAC operation.

There is a continuum of possible implementations for a dot-product, based on the tradeoff between delay and resources [Ma and Taylor, 1990]. Starting from the high delay, low resource end of the continuum, the major implementation points are:

1. Single serial multiplier (Section 7.3.1). Every clock cycle it generates and accumulates one partial product, not resetting the accumulator between products.

2. Single parallel multiplier (Section 7.3.2), plus an accumulator. Every clock cycle it generates N partial products with fixed shifts. These are summed with an adder tree and this result is accumulated with the results from other multiplications. This combination of multiplier and accumulator is commonly called a MAC unit. In a general purpose digital signal processor the state of the art for a parallel binary multiplier is to use Booth style recoding [Bewick, 1994]. Whilst this implicitly uses redundancy to halve the number of partial products and hence adders, it does not reduce the number of multiplications that must be performed sequentially.

3. Multiple parallel multipliers plus multiple accumulators. Every clock cycle each multiplier generates a product that is accumulated. The final dot-product result is generated by summing the multipliers' accumulators. Modern digital signal processors usually use this approach with two MAC units. Field programmable gate

arrays (FPGAs) often have many multipliers allowing dot-product multiplications to be apportioned over many MAC units to reduce the delay [Tatas et al., 2007, Altera Cyclone III, 2008, Xilinx Spartan-3, 2008].

4. Multiple constant multiplication (MCM) [Voronenko and Püschel, 2007]. The dot-product is broken down into its constituent partial products and all are added in parallel, similar to technique 3. This takes a large number of resources and is completed in a single clock cycle. As the coefficients of vector **A** are known in advance and will not change, the zero partial products can be ignored to both reduce the resource usage and decrease the delay [Boullis and Tisserand, 2005]. Minimum Hamming weight binary signed digit (BSD) can be used to increase the proportion of zero partial products [Kharrat et al., 2002, Wang and Roy, 2005]. Other techniques can be used to factorise the partial products within a multiplication, known as sub-expression sharing, to further reduce the additions [Koc and Hung, 1992, Dempster and Macleod, 2004].

This chapter discusses the hardware capable of taking advantage of the partial product packing (PPP) of Chapter 8. Its implementation is similar in architecture to a parallel multiplier, point 2 in the continuum above, although its operation can be interpreted as multiple serial multipliers without shifters, or though PPP as multiple parallel multipliers.

The optimal number of independent partial product units for implementation is investigated in Section 9.1. An FPGA based implementation is presented in Sections 9.2 through 9.4. The performance of this hardware is compared to a standard binary implementation in Section 9.5. Some variations on the use of the hardware design such as polyphase filter implementation are briefly discussed in Sections 9.6 through 9.7. Finally, a summary is given in Section 9.8.

9.1 DELAY – RESOURCE TRADEOFF

A balance is sought between the time taken to calculate the dot-product and the resources that it consumes. The continuum can be parametrised by the number of partial product units (PPUs) that are required. Assume that each implementation alternative can take advantage of minimum Hamming weight representations and avoid adding all zero partial products. If the word length is N and the vector length is L then there are NL partial products. Employing minimum Hamming weight BSD reduces the number of nonzero partial products to a third giving a total of $NL/3$ partial products to be accumulated, if the zero valued partial products are ignored [Koc and Johnson, 1994]. The serial multiplier uses a single PPU to generate and accumulate all dot-product

partial products, taking $NL/3$ clock cycles. The single parallel multiplier uses N PPU, taking $L/3$ clock cycles. MCM uses $NL/3$ PPU and a single clock cycle.

The optimal number of PPUs, M, depends on both the resource and performance criteria. Performance is usually dominated by the number of clock cycles, however as the number of PPUs increases, the number of levels in the adder tree must also increase, leading to an increase in the critical path propagation delay. Similarly, as the number of PPUs decreases, a larger barrel shifter is required which also lengthens the critical path delay. Therefore, the total calculation time for a dot-product operation can be estimated as the propagation delay though a shifter if required and the addition tree multiplied by the number of clock cycles required to process all partial products. The total resource usage can be estimated as the resources for all PPUs, including shifters, and the resources for the addition tree. The optimal number of PPUs is found where the product of the total time and total resources is minimised.

Taking into account the previous assumptions and design optimisations a simulation was developed to estimate the total time and resources required to calculate a dot-product for a given number of partial products. Simulations were run for $N = 32$ symbol representations and varying numbers of partial products. This generated curves for the product of total time and resource usage as shown in figure 9.1. The curves step down from the left as the shifter propagation delay decreases until the shifters need to only perform two shifts at $N/2 = 2^4 = 16$ PPUs. The curves step up to the right of $N = 2^5 = 32$ PPUs as the propagation delay increases again as more levels are required in the addition tree. The sawtooth shape results from some PPUs being left under-utilised. The optimum time-resource product is between $N/2$ and N PPUs, showing that a single parallel multiplier, with N PPUs, provides a good compromise between speed and resource usage.

9.2 MULTIPLY ACCUMULATE UNIT DESIGN WITH INDEPENDENT PARTIAL PRODUCT UNITS

The proposed MAC unit is designed with $M = N$ PPUs as indicated in Section 9.1. The architecture is like a standard parallel multiplier, where each partial product unit has a fixed shift, except the PPUs can generate any partial product from its packed row, independent of the other PPUs. The vector **B** is implemented as a delay line, suitable for implementing a finite impulse response (FIR) filter and supports binary data as the inputs. The vector **A** coefficients may be represented with a redundant number system and packed as in Chapter 8. The data output number system is binary signed digit (BSD) that is optionally converted to binary using a decoder (Section 6.2.1). This allows

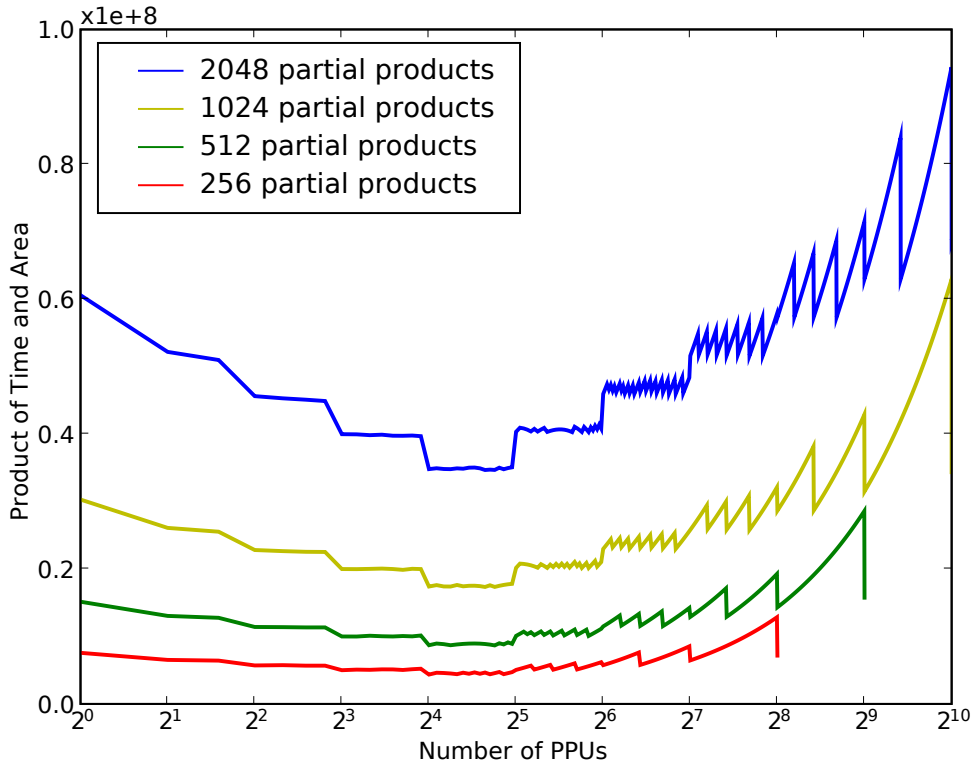


Figure 9.1: The product of the total dot-product calculation time and design area in terms of the number of PPUs implemented for coefficients with word lengths of 32 symbols.

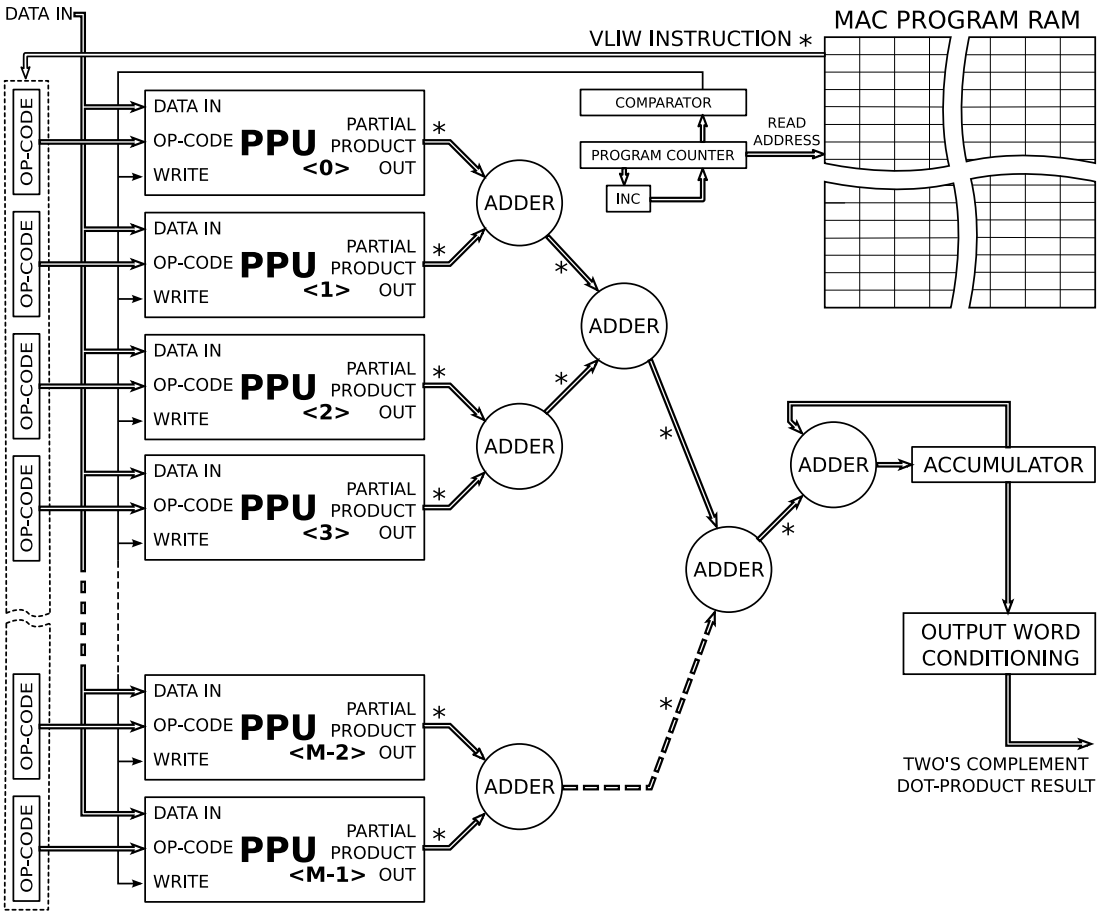


Figure 9.2: High level logic diagram of the proposed MAC implementation showing the logic connection but no timing. A $\xrightarrow{*}$ marks the position of a pipeline register.

it to be used as a drop in replacement for a MAC implemented with binary. Figure 9.2 shows the high level structure of the MAC unit. It consists of three major parts:

1. The control unit including a program random access memory (RAM) containing the vector **A** of coefficients appropriately rearranged and address counters.

2. A bank of PPUs to independently generate the partial products.

3. An adder tree to accumulate the partial products and produce the dot-product result.

The packed partial product sequence describing the dot-product is stored in the program RAM as very long instruction words (VLIW). One VLIW describes one column of partial products. The VLIW are unpacked into individual operational codes (op-codes) that instruct a PPU on which partial product to generate in its row. Each clock

cycle the program counter increments and a new column of partial products is generated. The program RAM has a depth of at least $R_{\max}$ VLIWs and a width of M op-codes.

To facilitate the PPP, each PPU needs random access into the filter delay line containing the vector **B** multiplicands. To implement this, each PPU maintains its own delay line memory. A single delay line memory shared by all PPUs would not suffice since M parallel read operations are needed. To keep their own copies of vector **B**, each PPU is connected to the binary DATA IN bus and WRITE strobe. With the dual-port block RAM on Spartan 3 and Cyclone III FPGAs, the write operation can occur in parallel to one of the read operations. Optionally, two PPUs can share one delay line RAM, however, this adds an extra clock cycle to the dot-product for the write operation that now must occur between reads.

The PPUs feed an adder tree that compresses the result into the accumulator. Upon completing the dot-product operation, the BSD result is converted into two's complement binary using the decoder of Section 6.2.1. The input and output interfaces use the standard two's complement binary number system allowing this MAC unit to be used as a drop in replacement for a binary MAC.

9.3 PARTIAL PRODUCT UNIT

The heart of the MAC unit design is the PPU. As shown in figure 9.3, each PPU is broken up into a circular random access delay line, an optional selectable shift unit and a partial product generator. The PPU generates a partial product based on the instruction of an op-code with the fields:

1. MULT SYM, a few bits for the multiplying symbol,
2. SHIFT, one optional bit for a selectable shift (Section 9.6), and
3. DELAY ADDRESS, $\lceil \log_2(L) \rceil$ bits for the index i of the multiplicand data $\mathbf{b}_i$ in the delay line.

9.3.1 Circular Random Access Delay Line

The circular delay line consists of a dual port RAM and address generation logic. The addressing for reading and writing is circular. The start of the ring buffer is specified by the base (zero delay) address, which is stored in a register. This is updated to the next base address after each write to the delay line.

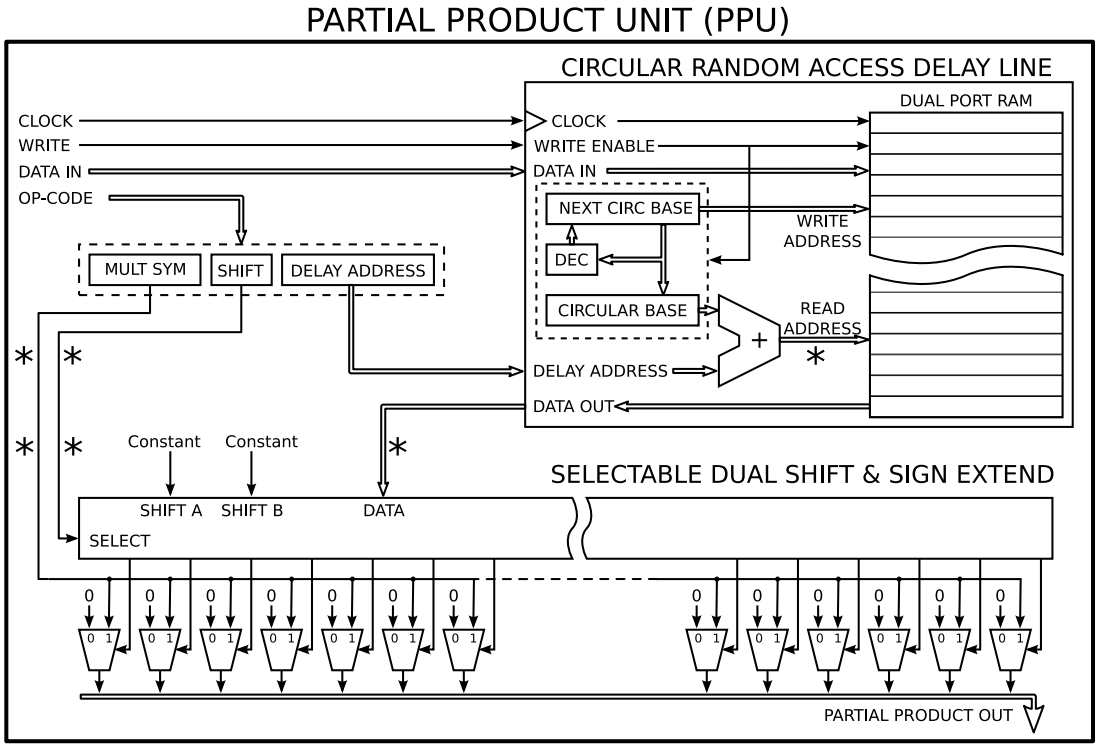


Figure 9.3: Logic diagram of a PPU with the optional selectable shifter. A $\xrightarrow{*}$ marks the position of a pipeline register.

The write address is the circular base address minus one. A write to the delay line RAM occurs on the rising clock edge when the WRITE strobe signal is high. This prompts:

1. The RAM to write the data on the DATA IN bus to the next circular base address.

2. The CIRCULAR BASE address register to be updated with the latest write address making it the address of zero delay.

3. The write address register, NEXT CIRC BASE, is decremented by one, ready for the next write cycle.

Two address registers are used to store the circular base and the write address. This removes the base decrementing operation from the critical delay path.

The read address is calculated as the DELAY ADDRESS plus the CIRCULAR BASE address. The RAM size is a power of two large enough to accommodate the delay line of length L. Being a power of two ensures that the read address will automatically wrap around to the start of the RAM block when the addition overflows.

9.3.2 Partial Product Generation

When the multiplicand has been retrieved from the delay line RAM it is left shifted and multiplied by the MULT SYMBOL from the op-code. The fixed left shift and binary two's complement sign extension requires no logic, so it is accomplished with hardwiring. An optional and selectable alternate shift maybe specified by the SHIFT bit of the op-code. If this is allowed then the additional shift is implemented as two hardwired shifts selected with a multiplexer as shown in Figure 9.3 and in more detail in Figure 7.6.

The optional shift and partial product generation can be amalgamated into a single level of LUTs as shown in Figure 7.6. This means there is no performance disadvantage in allowing the optional shift. Some extra LUTs will be used; the difference in the shift constants, to provide more multiplexers where the shifts do not overlap. However, as described later in Section 9.6, this may allow two rows to occupy the same hardware eliminating an entire PPU and adder, possibly reducing the depth of the adder tree.

The partial product is generated with multiplexers that either output a 0-symbol if the data at that position is a 0-bit, or output the multiplying symbol copied from MULT SYM in the op-code if the data is a 1-bit. This replaces all 1-bits in the binary data sample with the multiplying symbol and promotes all binary 0-bits to 0-symbols.

To generate a partial product the PPU performs the following operations on the rising clock edge:

1. Calculate the read address for the data as the op-code DELAY ADDRESS plus the CIRCULAR BASE address.
2. Read the data sample from the calculated address in the delay line.
3. Left shift the data sample by the amount specified by one of two constants, SHIFT A or SHIFT B, selected by the op-code SHIFT bit (if used).
4. Sign extend the data to the full length of the coefficient width plus data width.
5. Generate the partial product at each position with multiplexers that use the multiplicand bits to select a zero-symbol or MULT SYMBOL.

9.3.3 Adder Tree

When a selection of partial products have been generated by their respective PPUs they are added to the running accumulation. This is accomplished most quickly with a tree of redundant adders.

The redundant number system with $\mathbb{S} = \{\bar{1}, 0, 1, 3\}, \beta = 2$ is used for the experimental results presented in Section 9.5. To reduce the required resources, the adder of Section 5.6 is employed, with the restriction of disallowing the addition of two 3-symbols as discussed in Section 5.10.1. All other symbol additions are legal and it does not matter which addend has the 3-symbol. The 3+3 case can be avoided by rearranging the partial products so that two multiplying 3-symbols do not coincide on the neighbouring PPU that feeds the same adder. A simple strategy to ensure this be employed as part of the MAC program RAM compiling process; sort the rows of partial products so that all 3-symbol partial products are calculated first in one PPU and last in the neighbouring PPU.

9.4 PIPELINING

Adding pipelining to the design increases the thoughput of the MAC unit. This is achieved by partitioning the logic into smaller sections with fewer levels of combinational logic. The result of each section is stored after each clock cycle and fed to the next section at the start of the next clock cycle. This allows an increase in clock frequency because each section has a smaller combinational logic delay than the entire unpipelined design.

The design is partitioned into eight pipelined sections:

1. **IF**, instruction fetch; reads the very long instruction word (VLIW) from the program memory and increments the program counter. In parallel to the zeroth instruction fetch of a dot-product the data sample is written **WR** to the circular delay lines.
2. **DEC**, decode; calculates the address in the circular delay line.
3. **DF**, data fetch; reads the data from the circular delay line RAM.
4. **PPG**, partial product generation; create the partial product with appropriate multiplier symbol.
5. **AT0** – **ATX**, addition tree; calculate one level in the addition tree.
6. **ACC**, accumulate; add the result from the adder tree to the running accumulation. For the zeroth accumulation of a dot-product the accumulator is zeroed and the previous result is sent to the binary decoder (**BIN**).

The pipeline register positions are marked with a $\xrightarrow{*}$ in Figure 9.2 and Figure 9.3. The pipeline execution schedule is shown in Figure 9.4.

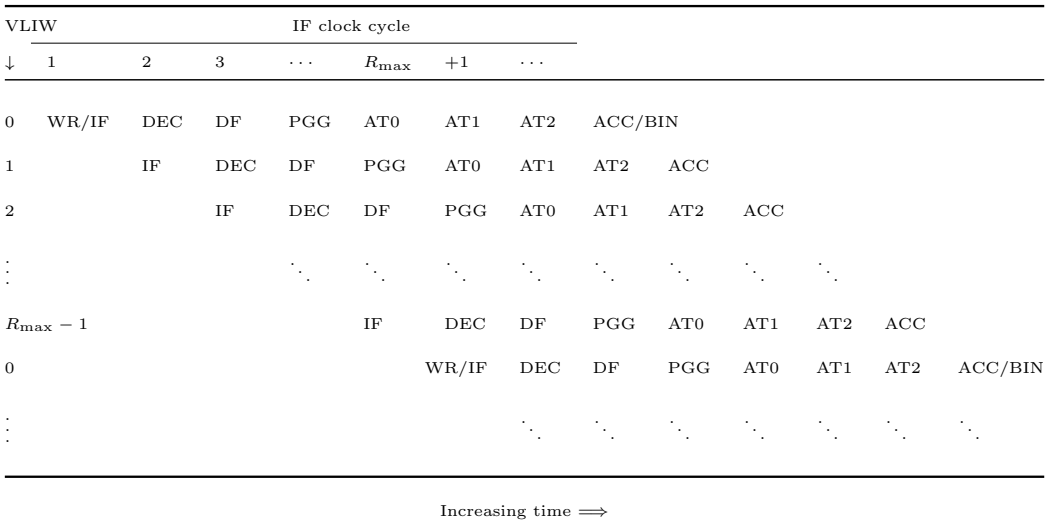


Figure 9.4: Pipeline diagram showing the execution order of the pipeline stages. While the zeroth VLIW is decoding (DEC) the next VLIW is being fetched. After $R_{\max}$ clock cycles the zeroth VLIW is fetched again for the next dot-product. The delay line write (WR) occurs in parallel with the zeroth VLIW IF. The binary decode (BIN) starts with the zeroth VLIW accumulate and continues in parallel with the rest of the pipeline.

The binary decoder runs in parallel with the rest of the pipeline at a rate $R_{\max}$ times slower than the VLIW instruction execution since the accumulator result is only valid every $R_{\max}$ cycles. This gives ample time for the relatively slow binary decoder to execute.

9.5 PERFORMANCE COMPARISON

A 32-bit, 51 tap low-pass FIR filter was used as a test case for the performance of the proposed dot-product design. To minimise the Hamming weight of the tap coefficients and thus maximise the packing the $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$, $\beta = 2$, $N = 32$ number system was used with the PPP optimisation of Section 8.6. This resulted in a relatively poor maximum row packing of $R_{\max} = 17$, being a lower half result according to Figure 8.8. Eight pipeline stages were used.

The proposed architecture was compared to two binary implementations, using binary multipliers. A four multiplier design was chosen for comparison so that the number of cycles required, $51/4 = 13$ is better than the best achievable PPP result according to Figure 8.8. The multiplication results were summed with a tree of ripple-carry adders and accumulated with a ripple-carry adder. The first binary implementation used the dedicated 9-bit multipliers in the FPGA. The second used general logic to

Table 9.1: Synthesis results for 32-bit dot-product implementations of a 51 tap FIR filter in a C6 speed Altera Cyclone II.

	$\{\bar{1}, 0, 1, 3\}$	Binary 1	Binary 2
Maximum Frequency (MHz)	220	145	112
Pipeline Latency (cycles)	8	8	8
9-bit Multipliers	0	32	0
Total Logic Elements	9099	1289	5588
- Combinational Functions	8744	828	4899
- Dedicated Registers	4299	1270	4923
Memory Bits	57696	8192	8264
MAC cycles	17	13	13
Data throughput (MSPS)	**12.9**	**11.2**	**8.6**

implement the multipliers. Each binary implementation was allowed to use eight pipeline stages and the register re-timing feature of the synthesis software. The designs were compiled using Altera Quartus 6.1 with the highest speed grade (C6) Cyclone II FPGA as the target device.

The implementation results are presented in Table 9.1. Although the number of cycles to calculate the dot-product is greater for the proposed implementation with PPP, its higher operational frequency gives a lesser delay per MAC cycle, compensating to give a higher throughput.

9.6 HARDWARE ENABLED OPTIMISATION

The number of PPUs can be manipulated to achieve a reduction in resource usage or an increase in performance, as illustrated in Section 9.1. A PPU can generate partial products with two different shifts using the optional SHIFT bit in the op-code as described in Section 9.3. This would combine two rows for execution in a single PPU; reducing the resource usage. Conversely, two PPUs can be specified with the same fixed shift. This would split a single row across two PPUs.

As explained in Section 8.7.1, the zeroth row's length is irreducible and is likely to be longer than the other rows, up to twice as long. Using row splitting, the partial products of the zeroth row are apportioned over two PPUs with a fixed shift of zero. This gives, a large gain in time as the number of clock cycles has been reduced by $R_{\max} - \lceil R_0/2 \rceil$, where R_0 is the length of the zeroth row. This time improvement costs only the resources for an addition PPU.

Row combining can be used to reduce resource usage by amalgamating two short rows and employing the optional dual shifter. If the combined length of the two rows is less than maximum row length $R_{\max}$ then no performance is lost. Performance may be gained if the adder tree requires fewer levels. This can be used for the more significant rows when implementing a FIR filter. The more significant rows are shorter as there are only a few large values in a FIR filter impulse response.

9.7 SUPPORT AND OPTIMISATION CONSTRAINTS FOR FILTER VARIATION

The use of constant coefficients for vector **A** was assumed throughout the development of the optimised dot-product MAC unit. However, the design process showed that much of the performance advantage can be captured in the MAC program RAM. Therefore, the same general hardware implementation can be used for the calculation of multiple filters by modifying the program RAM contents. This is applicable where the coefficients may change during execution, for example, block adaptive filters [Mikhael and Wu, 1987, Ogunfunmi and Peterson, 1989], or channel selection filters in a software defined radio [Blossom, 2004]. In these cases, to exploit the performance advantages of PPP, the optimisation process must be included after the adaption process.

Multi-rate processing is used to change the sample rate of a signal, down by decimation, or up by interpolation. The purpose is to resample the signal or perform some intermediate processing at a slower rate as shown in Figure 9.5. Polyphase filters combine decimation or interpolation of the digital signal with the appropriate anti-aliasing FIR filters [Ifeachor and Jervis, 2002]. Polyphase filter implementations ignore certain samples of the filter input stream, which lets them operate at the lower sample rate. The anti-aliasing filter before down-sampling does not need to calculate the samples which are dropped, so it is split into several smaller filters with their results combined and output at the lower rate, as shown in Figure 9.6. Likewise, the interpolation filter does not need to use the zero samples inserted by the up-sampling. It also breaks the interpolation filter into several smaller filters as shown in Figure 9.7. By avoiding the multiplications associated with the samples known to be zero, the calculation effort is reduced. In terms of implementation, this divides the antialiasing and interpolating FIR filters into several smaller sub-filters, depending on the up- or down-sampling rates. These sub-filters are used in a round robin to generate the output samples, with an equivalent impulse response to the original filter.

The PPUs' random access into the delay line makes it possible to encode many small filters and write a filter program RAM that calculates them sequentially. Only the

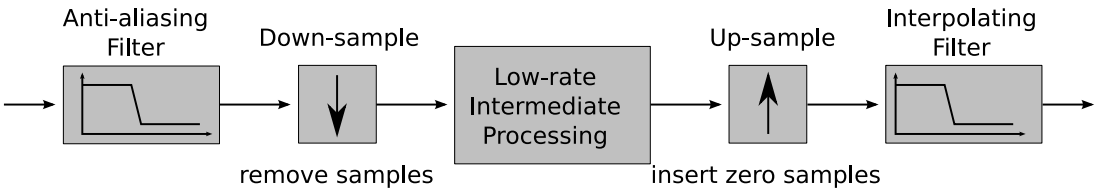


Figure 9.5: Multirate processing chain. The sample rates on the left of the down sampling, to the right of the up-sampling and in between can all be at different sampling rates.

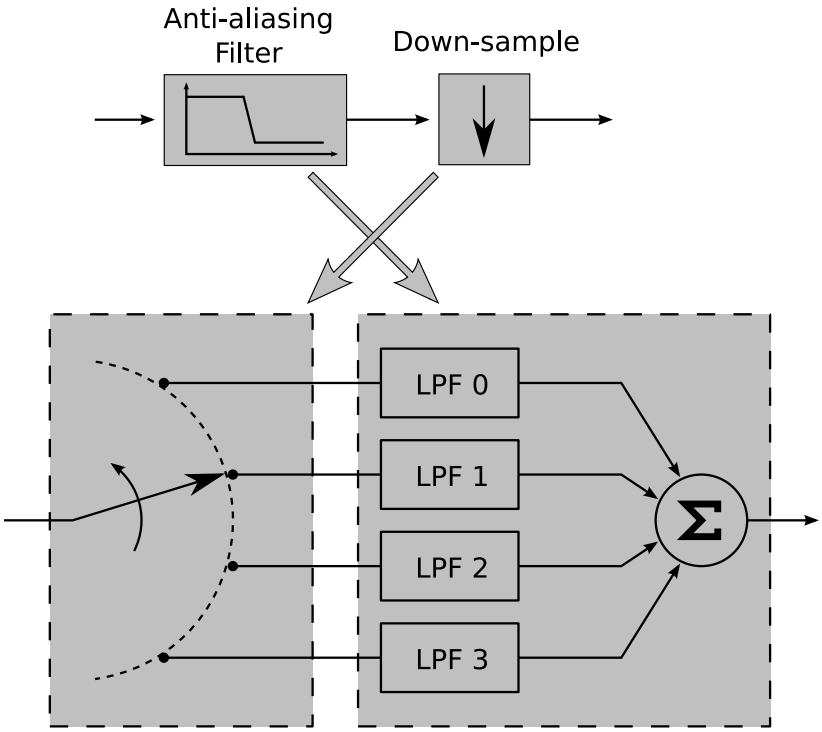


Figure 9.6: Polyphase decimation filter implementation for down-sampling by four. As a sample enters the left it is inserted into one of the low pass filters (LPF). When each filter has a new sample their outputs are summed to give an output sample.

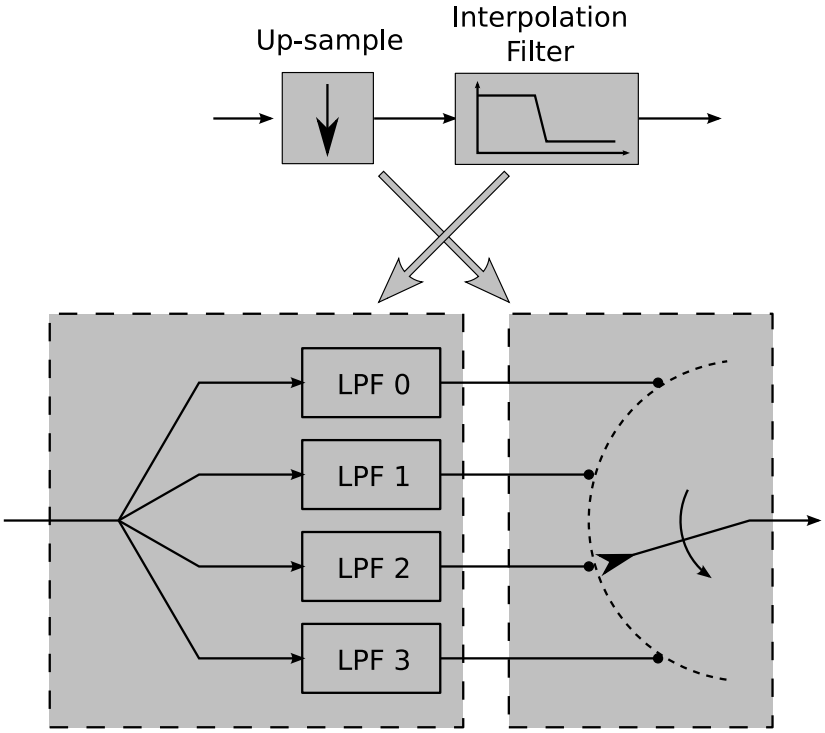


Figure 9.7: Polyphase interpolation filter implementation for up-sampling by four. Every input sample is given to each low pass filter (LPF). For each input, four results are read to give four new samples for the output stream.

timing of the reading in and writing out of samples needs to be changed. Each sub-filter would be packed and optimised individually, with the start of each filter corresponding to the zeroing of the accumulator.

9.8 SUMMARY

The proposed implementation of a MAC unit presented in this chapter facilitates the realisation of the PPP of Chapter 8. The core concept is the independently generating partial products at each fixed shift. This allows the logic to continuously perform useful calculations with the significant partial products, rather than wasting time on zero partial products that can be ignored.

The design primarily uses $M = N + 1$ PPUs in parallel, where two PPUs are used for the least significant shift and each of the remaining PPUs have a fixed shift between 1 and $N - 1$. It was shown that M, the number of PPU, should be in the range $N/2 \leq M \leq N$ where the product of time and resource usage is at a minimum. The number of PPUs instantiated can be reduced by including a variable shift operation within a PPU so that it can generate the partial products of two rows. Including this shift does not increase the resources or delay in an FPGA implementation. This row combining can be used after the optimised PPP when two shorter rows exist, such as with the most significant rows of a FIR filter. If the sum of the row lengths is less than the maximum row length $R_{\max}$ then this does not increase the time and does decrease the resource usage.

The presented implementation is faster than an equivalent binary implementation without PPP. An implementation for a 51-tap 32-bit FIR filter was synthesised. After PPP the filter had an $R_{\max} = 17$. Combined with the constant time addition, it gave a data throughput of 12.9 million samples per second (MSPS) at a clock rate of 220 MHz. This was 1.7 MSPS (15%) faster than a 145 MHZ binary implementation with four multipliers even though it had an advantage of four fewer clock cycles; an equivalent $R_{\max}$ of $\lceil 51/4 \rceil = 13$.

The PPP MAC implementation presented in Section 9.2 is more flexible than a multiple-constant-multiplication dot-product implementation. The coefficients of vector **A** can be changed by modifying the program RAM. It can also be used to implement polyphase filter banks, with the samples of vector **B** being updated more often.

Chapter 10

ALPHABET SELECTION

The space of possible number systems is infinite. Even when extreme radix and symbol alphabets are ignored there are still a huge number of combinations. This chapter considers some tight bounds that can be placed on the values of the radix and symbol alphabet such that their implementation is practical in current technology. The maximum cardinality of the alphabet is also limited to be practical in terms of the logic and wiring resources consumed. Some number systems satisfying these constraints are explored, evaluating the minimum Hamming weights in their average and worst cases.

Section 10.1 describes how the implementing technology, for instance field programmable gate arrays (FPGAs) with 4-input look-up tables, restricts the cardinality of the symbol alphabet. The symbols that should be considered for inclusion are detailed in Section 10.2. The optimal radix and practical limits on the radix are discussed in Section 10.3. The Hamming weights of representations in a number system are of importance in algorithms employing multiplication. The average and worst case minimum Hamming weights for various radix-2 number systems are calculated and discussed in Sections 10.4 and 10.5 respectively. The results from these sections demonstrate how the symbol choice can affect the minimum Hamming weights of a number system. Section 10.6 discusses the hardware cost of having large symbols in the alphabet. Finally, a summary is given in Section 10.7.

10.1 PRACTICAL LIMITS ON SYMBOL ALPHABET CARDINALITY

The target technology and in particular the cost of implementation on that technology is the limiting factor for the number of distinct symbols that can be employed. The dominant technology for integrated circuits is complementary metal-oxide-semiconductor (CMOS) which is used to build fast, low power and compact digital circuits with *two* reliably stable output states [Chandrakasan et al., 1992]. These output states are

labelled 'high' and 'low', or H and L, or 1 and 0. The binary number system dominates in digital circuits because the cardinality of each bit matches the two stable states of a signal. Some work has been done towards ternary CMOS gates with three stable signal states [Gundersen et al., 2005]. However, these gates occupy considerably more silicon area, consume more power, and are slower to switch states than an equivalent circuit with two stable states. Silicon on insulator (SOI) integrated circuit technology can provide three stable states since its bulk substrate is an insulator and hence is not connected to ground [Beckett, 2009]. This allows circuits which can pull a signal wire to a positive or a negative voltage or sit at zero voltage, with the same efficiency and similar area to CMOS circuits. Unfortunately this technology is not yet commercially available.

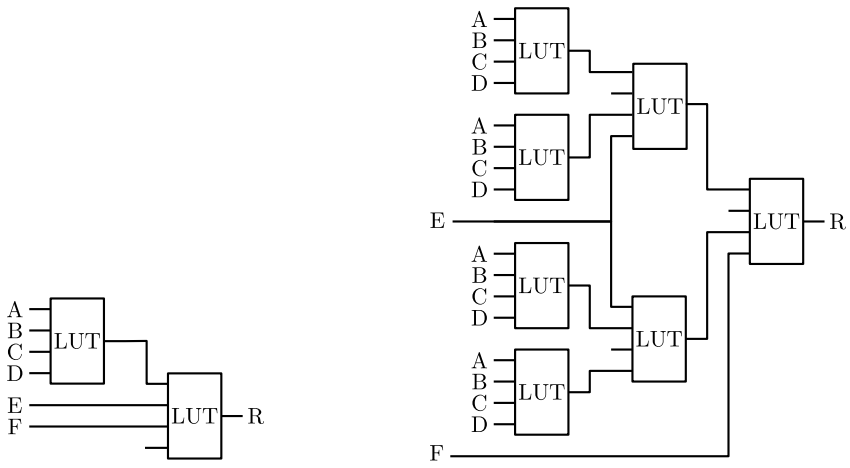
As discussed in Section 2.3, a positional number system must have a radix with a magnitude greater than or equal to two. To have redundant representations, a number system must have a symbol alphabet with cardinality greater than the radix, i.e., a minimum cardinality of three for the minimum radix of two. However, to describe the three symbols requires more than the two digital states of a signal. The combinations of at least two digital signals must be employed to represent the three or more symbols of a redundant alphabet.

The number of possible ordered state combinations for k digital signals is,

$$c = 2^k. \tag{10.1}$$

As k is increased, the number of encodable symbols and thus the possible cardinality of the symbol alphabet increases exponentially. However, the cost of computing with larger symbol alphabets also increases exponentially. An increase in the number of input signals causes the number of gates to increase at somewhere between a linear rate, for the simplest functions, and an exponential rate, for worst case functions. This is discussed further in Appendix C for 4-input look-up tables (4-LUTs).

Consider an arbitrary operation that takes two symbols as input and outputs a third, for example, selecting the maximum of the two input symbols. Since a 4-LUT can implement any four-to-one logic function it can accommodate the logic for any operation on a pair of two-signal symbols. Thus, only two 4-LUTs are required to generate the two signals of the result symbol. However, if three signals per symbol are used, then this same function on two symbols totals 6 inputs signals and three output signals. Depending on the complexity of the function, its implementation with 4-LUTs requires three cones of logic of at least two 4-LUTs each and up to seven 4-LUTs according to Figure 10.1. For three output signals this totals between $3 \times 2 = 6$ and $3 \times 7 = 21$ 4-LUTs; considerably more than the two 4-LUTs need for two signal symbols.



(a) Best case. (b) Worst case.

Figure 10.1: The two extremes on the number of 4-LUTs required to implement an arbitrary 6 input logic function. See Appendix C for further explanation.

This inflation of resources puts a practical limit on the alphabet of four symbols when implementing in an FPGA with 4-LUTs.

For an FPGA with 6-input LUTs this limit grows to allow $k = 3$ signals for each symbol, giving a usage of three 6-LUTs for a similar function. Keep in mind that a 6-LUT consumes at least four times the silicon area of a 4-LUT so the three 6-LUTs have approximately the equivalent silicon area of twelve 4-LUTs.

An application specific integrated circuit (ASIC) is not typically built using LUTs, because the logic is more fine-grained; individual gates are used. The gate 'fan-in' is roughly equivalent to the number of inputs to a LUT for this argument. The maximum fan-in of a CMOS gate is dependant on many factors, including the CMOS technology, the supply voltage, and required switching speed. Nevertheless, the same principle that an increased number of signals per symbol increases the circuit area still applies. In general, the silicon area for an ASIC will be less than that used for an FPGA, but both grow exponentially with the number of input signals to the logic functions.

10.2 OTHER SYMBOL ALPHABET CONSIDERATIONS

When selecting a symbol alphabet for a specific application there are a few other constraints to consider. For instance, how will the zero-symbol be used? What range of values are required? Are signed values needed? Will comparison be required? These influence the uses of the number representations, their range, and their Hamming weight.

10.2.1 Zero-symbol

The symbol alphabet should have a zero-symbol. Without it, the Hamming weight and zero-dominant properties cannot exist. The performance exploits associated with generating zero partial products in multiplication (Section 7.2), or addition with zero-symbols (Section 7.3) cannot be employed to speed up some algorithms. The zero symbol provides a large portion of the benefit derived from redundant number systems and so it should always be included in the symbol alphabet.

10.2.2 Continuous range

A number system with a large continuous range, as described in Section 2.5, is usually desirable. The simplest way to ensure this is to include all the non-redundant symbols in the symbol alphabet. For radix-2 systems, a continuous positive range of at least $[0, 2^N)$ is ensured by including the binary symbols 0 and 1. Each value then has at the least its binary representation.

Including the non-redundant symbols also makes interfacing to non-redundant systems easier, especially if the encoding of these symbols is consistent between the two systems. For example, including the symbols 0 and 1 in a radix-2 redundant number system where the encoding for these two symbols only differs by one bit. This reduces conversion from binary to the new system the inclusion of additional bits to encode the other symbols.

10.2.3 Negative symbols

Negative symbols allows negative numbers to be treated identically to positive numbers. The value's sign is incorporated into the representations' symbols. Using the sign detection method of Section 6.3, the sign of a representation can be detected in $O(\log(N))$ time and $O(\log(N))$ resources.

Without at least one negative symbol, negative values require special treatment in hardware. The sign-magnitude solution requires extra logic to detect a change in sign and to modify a sign bit. With redundant symbols this can be costly to implement as it is difficult to detect the magnitude of a representation without decoding it to a non-redundant form, see Section 6.4. It is best to avoid this conversion because it is slow, see Section 6.2.

10.2.4 Unique representation for zero value

To use the sign and zero detection circuit of Section 6.3, the value of zero needs to have the unique representation $000\cdots000_\beta$. This can only be ensured if, for every permutation of two symbols s_1 and s_2 from the symbol alphabet, the following condition holds,

$$s_1 \neq -\beta^n s_2, \qquad \forall \quad n \geq 1, \quad \forall \quad s_1, s_2 \in \mathbb{S}, \tag{10.2}$$

except when $s_1 = s_2 = 0$. Failing this condition with $n = 1$ means that zero can have the repeating representation $(s_2 s_1)\cdots(s_2 s_1)_\beta$. Critically, evaluating the group $(s_2 s_1)_\beta$ with (2.1) gives,

$$s_2\beta^{i+1} + s_1\beta^i = 0. \tag{10.3}$$

For example, the number system $\mathbb{S} = \{\overline{1}, 0, 2\}$,$\beta = 2$ fails the condition of (10.2) since, $2 = -2^1 \times \overline{1}$. This allows a representation for zero of $\overline{1}2\overline{1}2\cdots\overline{1}2\overline{1}2_2$.

10.3 PRACTICAL LIMITS ON THE RADIX

The radix plays an important role in a number system and its implementation. With a greater radix the same value can be represented with fewer positions. However, a larger symbol alphabet is required, with a minimum of $\beta + 1$ symbols for a redundant number system as discussed in Section 10.1. The larger alphabet also means an increase in complexity.

A radix of three is promoted in the article by Hayes [2001] because it is closest to the natural exponent $e = 2.71828183\cdots$. A radix of two comes a close second. It is proposed that the product βN is indicative of the hardware required to implement a non-redundant number system and its operations. As the radix, β increases, the word length N decreases for a fixed range of values. A radix of three minimises the hardware cost product βN and therefore, is the most resource efficient. Occasionally, at small ranges, a radix of two is slightly better. This result assumes a non-redundant number system where $|\mathbb{S}| = \beta$. To account for redundant number systems with $|\mathbb{S}| > \beta$ the product $|\mathbb{S}|N$ should be minimised. The radix no longer features in the optimisation product, however, the previous result for non-redundant number systems indicates that the radix should be small.

Assuming the use of 4-LUTs for implementation, the practical limit on the symbol alphabet cardinality is four, as described in Section 10.1. This limits the radix to $2 \leq \beta \leq 4$ or $-4 \leq \beta \leq -2$. If a redundant number system is desired then only a radix of magnitude two or three will work with a four symbol alphabet. Hence, the practical

limits on the symbol alphabet restrict valid integer radii to -3, -2, 2, or 3.

10.4 AVERAGE MINIMUM HAMMING WEIGHT

The average minimum Hamming weight of a number system is an important property. Representations with minimum Hamming weight reduce the number of nonzero partial products in a multiplication. With the optimised dot-product packing of Section 8.6, the number of effective multiplications is approximately proportional to the average minimum Hamming weight of the number system employed. To maximally exploit this packing the number system should have a low average minimum Hamming weight.

A method of generating minimum Hamming weight representations for any number system definition is described in Section 3.2. This method developed a Mealy output finite state machine. The Markov analysis of the state machine in Sections 3.3 and 3.4 computed an expression for the average minimum Hamming weight of representations described by the state machine and thus those expected for the number system.

The minimum Hamming weight of some number systems were compared by generating their corresponding minimum Hamming weight encoding state machines. The number system definitions were restricted to symbol alphabets of up to four symbols and a radix of two, in keeping with Sections 10.1 and 10.3. Each alphabet's ability to recode a representation to maximise the number of zeros was found by performing a Markov analysis on its recoding state machine. The results of these trials are presented in Table 10.1.

From the number systems tested in Table 10.1, the symbol alphabet with the lowest average minimum Hamming weight is $\{\overline{3}, 0, 1, 7\}$, although a window width of $m = 9$ or more is required to achieve this. Recall that the window width is the number of input bits that the state machine uses to decide which state to output. It is indicative of the size of the state machine. The symbol alphabets of $\{\overline{1}, 0, 1, 3\}$, $\{\overline{3}, \overline{1}, 0, 1\}$, and $\{0, 1, 3, 5\}$ have a similar average minimum Hamming weight to $\{\overline{3}, 0, 1, 7\}$ but only require a window width of 3. Consequentially, their state machines will be much smaller. The tradeoff between minimum Hamming weight and state machine size should be considered if the state machine is implemented and used in the digital signal processing (DSP) system at runtime.

Symbols that are a radix multiple of another symbol do not provide any additional reduction to the average minimum Hamming weight. This can be seen by comparing the average minimum Hamming weight of the symbol alphabets $\{0, 1, 2\}$ and $\{0, 1, 2, 4\}$ to that of binary $\{0, 1\}$ where 2 and 4 are radix-2 multiples of 1. Another example

Table 10.1: The average minimum Hamming weight $\widehat{\mathrm{H}_{\min}(V)}$ for some symbol alphabets in radix-2. The window width required is m and has a maximum window width of $M = 9$. References are given for entries confirmed by other published results.

$\mathbb{S}$	m	M	$\widehat{\mathrm{H}_{\min}(V)}$	Confirmed in
$\{0, 1\}$	1	9	$0.5N$	
$\{0, 1, 2\}$	1	9	$0.5N$	
$\{0, 1, 2, 3\}$	2	9	$0.333333N + 0.111111$	
$\{0, 1, 2, 4\}$	1	9	$0.5N$	
$\{\overline{1}, 0, 1\}$	2	9	$0.333333N + 0.111111$	[Hartley, 1996, Arno and Wheeler, 1993]
$\{\overline{1}, 0, 1, 3\}$	3	9	$0.285714N + 0.163265$	[Phillips and Burgess, 2004]
$\{\overline{1}, 0, 1, 5\}$	8+	9	$0.273476N + 0.227644$	
$\{\overline{1}, 0, 1, 7\}$	8+	9	$0.275738N + 0.296868$	
$\{\overline{3}, \overline{1}, 0, 1\}$	3	9	$0.285714N + 0.163265$	[Phillips and Burgess, 2004]
$\{\overline{3}, 0, 1, 3\}$	5	9	$0.277778N + 0.209877$	
$\{\overline{3}, 0, 1, 5\}$	6	9	$0.268757N + 0.421665$	
$\{\overline{3}, 0, 1, 7\}$	9+	9	$0.263881N + 0.319966$	
$\{0, 1, 3\}$	2	9	$0.333333N + 0.111111$	
$\{0, 1, 3, 6\}$	2	9	$0.333333N + 0.111111$	
$\{0, 1, 5\}$	8+	9	$0.323212N + 0.276668$	
$\{0, 1, 3, 5\}$	3	9	$0.285714N + 0.163265$	[Phillips and Burgess, 2004]
$\{0, 1, 3, 7\}$	5	9	$0.277778N + 0.195988$	

is the symbol alphabets $\{0, 1, 3\}$ and $\{0, 1, 3, 6\}$, where 6 is a radix-2 multiple of 3. This behaviour is expected, since the symbols that are radix multiples represent the same input patterns as the non-multiple symbol. The larger symbol provides the same recoding, just in a lower position. For example, recoding 01111010_2 with the BSD alphabet $\{\overline{1}, 0, 1\}$ gives $1000\overline{1}010_2$ and with $\{\overline{1}, 0, 1, 2\}$ gives $0200\overline{1}002$ as a possible minimum Hamming weight representation. These have equivalent Hamming weight since the BSD representation simply has its instances of the pattern 10_2 replaced by 02_2. No Hamming weight advantage has been gained by the inclusion of the 2-symbol. Therefore, alphabets should have symbols that are co-prime to the radix in order to achieve a reduction in the average minimum Hamming weight.

In Table 10.1 the minimum window width of the number system $\{\overline{1}, 0, 1, 5\}$, $\beta = 2$ was not found. It must exceed the maximum window width used for the search of $M = 9$. An improvement in average minimum Hamming weight is observed as the maximum window width is increased as shown in Figure 10.2. However, the improvement suffers diminishing returns, with little advantage gained beyond a maximum window width of $M = 5$, where $m = 4$ for the symbol alphabet $\{\overline{1}, 0, 1, 5\}$. An increase in window width increases the complexity of the state machine exponentially, following the size of the recoding table, $O(\beta^m)$. This is also the case for several other alphabets as indicated by a + in the m column of Table 10.1.

Interestingly, the alphabets $\{\overline{1}, 0, 1\}$ and $\{0, 1, 3\}$ have the same average minimum Hamming weight of $N/3 + 1/9$. However, they have drastically different representations and they achieve different minimum Hamming weights for various values, but on average they are equivalent. For example, the binary representation 01111111_2 can be encoded as $1000000\overline{1}_2$ ($\mathbb{S} = \{\overline{1}, 0, 1\}$) with a Hamming weight of two clearly beating the encoded representation 01030303_2 ($\mathbb{S} = \{0, 1, 3\}$) with a Hamming weight of four. A counterexample is the recoding of binary 00110011_2 as $010\overline{1}010\overline{1}_2$ or 00030003_2. The minimum Hamming weight of the two alphabets differ when handling the binary patterns 11_2 and $01\ldots1111_2$. Combining the two alphabets gives $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$ which inherits the benefits of both and achieves an average minimum Hamming weight of $2N/7 + 8/49$ as shown in Table 10.1.

10.5 WORST CASE MINIMUM HAMMING WEIGHT

Often a design is limited largely by the worst case situation. The worst case minimum Hamming weight for a number system is the minimum Hamming weight representation with the most nonzero-symbols. In a multiplication this translates to the maximum number of nonzero partial products that must be accommodated.

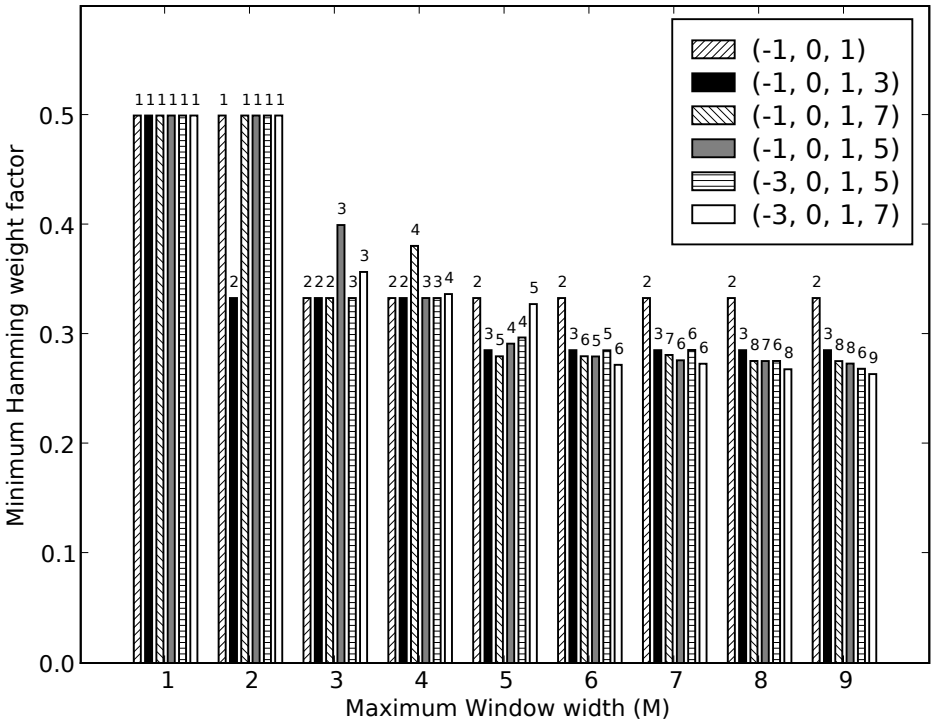


Figure 10.2: A comparison of various number systems' average minimum Hamming weight linear factor as the maximum window width M is increased. This allows further Hamming weight minimisation cases to be found. The window width m is given above each bar.

Considering the minimum Hamming weight recoding state machine of Section 3.2.3, the worst case minimum Hamming weight representation can be found from the cycle with the greatest ratio of Hamming weight to length. A cycle has a ratio,

$$r_{\text{cycle}} = \frac{w_{\text{cycle}}}{l_{\text{cycle}}}, \tag{10.4}$$

where w_{cycle} is the count of nonzero symbols output from the cycle's transitions, and l_{cycle} is the count of transitions in the cycle. Continuously traversing this cycle will output a representation with a weight of $r_{\text{cycle}}N$ as $N \to \infty$.

The recoding state machines are directed graphs and the list of elementary cycles within can be found using the algorithm of Johnson [1975]. Elementary cycles are paths through a graph that start and end at the same node and do not visit any other node more than once. The basic cycle search algorithm takes the following steps:

1. Select a starting node in the graph.
2. Recursively traverse the graph until a node in the current traverse is re-visited; a cycle has now been found.
   (a) If the re-visited node is the start node, report an elementary cycle by retracing the transitions.
   (b) Otherwise, ignore this cycle; it will be found again later.
3. Undo the recursion to find and explore a transition that has not yet been traversed.
4. When all transitions have been explored, select a new starting node and iterate.

Johnson's algorithm improves on brute force searches by trimming the search space and by only reporting a cycle once. This is achieved in two ways. Firstly, the start state is removed from the state machine after all the cycles through that state are found. At the next iteration a smaller state machine is searched. Secondly, cycles can only exist in strongly connected components of the state machine. A strongly connected component is a group of states where each state can be reached from every other state in the group. Searches for cycles are restricted to these strongly connected components which are found using Tarjan's algorithm [Tarjan, 1971]. As Johnson's algorithm iterates, a state is removed changing the strong-connectedness of the graph. The removed state may alienate some states as either unreachable or dead ends. These states can no longer form cycles and so are also removed.

The worst case cycle ratio is that with the largest r_{cycle}. Table 10.2 lists some radix-2 number systems and their worst case minimum Hamming weights. Many of the

Table 10.2: The worst case minimum Hamming weight for some symbol alphabets in radix-2. Maximum window width $M = 9$.

$\mathbb{S}$	m	M	Worst case r_{cycle}	Worst case output example
$\{0,1\}$	1	9	1	$\cdots(1)\cdots_2$
$\{0,1,2\}$	1	9	1	$\cdots(1)\cdots_2$
$\{0,1,2,3\}$	2	9	1/2	$\cdots(01)\cdots_2$
$\{0,1,2,4\}$	1	9	1	$\cdots(1)\cdots_2$
$\{\overline{1},0,1\}$	2	9	1/2	$\cdots(01)\cdots_2$
$\{\overline{1},0,1,3\}$	3	9	1/2	$\cdots(03)\cdots_2$
$\{\overline{1},0,1,5\}$	5+	6	4/9	$\cdots(0\overline{1}000\overline{11}05)\cdots_2$
$\{\overline{1},0,1,7\}$	6+	6		$\cdots()\cdots_2$
$\{\overline{3},\overline{1},0,1\}$	3	9	1/2	$\cdots(0\overline{1})\cdots_2$
$\{0,1,3\}$	2	8	1/2	$\cdots(03)\cdots_2$
$\{0,1,3,6\}$	2	8	1/2	$\cdots(03)\cdots_2$
$\{0,1,3,5\}$	3	8	1/2	$\cdots(03)\cdots_2$
$\{0,1,3,7\}$	5	8	2/5	$\cdots(00077)\cdots_2$

results are as expected. For binary the worst case ratio is 1, meaning that a cycle exists where every transition outputs a nonzero-bit. This corresponds to the representation $111\cdots111_2 = 2^N - 1$. For binary signed digit (BSD) the worst case cycle ratio is 1/2. This is corroborated by the canonical signed digit (CSD) representations that have the BSD alphabet $\mathbb{S} = \{\overline{1}, 0, 1\}$ and also have at least one zero-symbol between every nonzero-symbol [Hwang, 1979]. Hence, a worst case representation might be $01(01)\cdots(01)01_2$ where half the symbols are nonzero. The symbol alphabet $\mathbb{S} = \{\overline{1}, 0, 1, 5\}$ achieves a lesser worst case by recoding this case, for example the representation 1010101_2 becomes 0050005_2. In general, to have a worst case of less than a half, a symbol alphabet with a cardinality of greater than four is required, as shown by $\mathbb{S} = \{0, 1, 3, 5, 7\}$ and $\mathbb{S} = \{\overline{1}, 0, 1, 3, 5\}$ in Table 10.2. These alphabets provide recoding to one symbol for all three bit-patterns right shifted to remove the least significant zero-bits.

10.6 PRACTICAL LIMITS ON SYMBOL VALUES

A symbol's value can be anything, though larger symbols tend to make arithmetic more difficult and resource intensive. As a symbol's magnitude increases, it impacts on increasingly higher positions. This can be desirable when minimising the Hamming weight, however, in an adder it has consequences for resource usage and performance. With larger symbols the intermediate sum set will be large, since there is less opportunity for overlap, when two different symbol sums give the same intermediate sum value.

Consequentially a larger number of signed bits are required, creating an adder with more 3:2 compressors and greater depth. The signed bits after the first recoding level must initially carry further up the word. This makes the routing between the first and second layers longer, increasing routing delay. Hence, larger symbols will degrade both the resource usage and performance of adders.

10.6.1 Adder Sizes

The cardinality of the intermediate sum alphabet $|\mathbb{S}_\mathrm{I}|$ has the upper bound,

$$|\mathbb{S}_\mathrm{I}| \leq \frac{(|\mathbb{S}| + h - 1)!}{h!(|\mathbb{S}| - 1)!}, \tag{10.5}$$

where h is the number of symbols added to generate the intermediate sum and each symbol is assumed to have the same alphabet cardinality $|\mathbb{S}|$. This expression is the number of combinations of h symbols from the alphabet $\mathbb{S}$ with replacement. Note that the order of addition does not matter. Usually h will be two, giving the simplified expression,

$$|\mathbb{S}_\mathrm{I}| \leq \sum_{i=1}^{|\mathbb{S}|} i = \frac{|\mathbb{S}|\,(|\mathbb{S}| + 1)}{2}, \text{ when } h = 2. \tag{10.6}$$

The actual intermediate-alphabet cardinality may be less as a few of the intermediate sums may overlap. For example, in the addition of two symbols, $\{\overline{1}, 0, 1, 2\} + \{\overline{1}, 0, 1, 2\} = \{\overline{2}, \overline{1}, 0, 1, 2, 3, 4\}$, the intermediate sum symbols 0 ($= 0 + 0$, or $= \overline{1} + 1$), 1 ($= 0 + 1$, or $= \overline{1} + 2$), and 2 ($= 0 + 2$, or $= 1 + 1$) can be generated by different symbol sums.

With a symbol alphabet that is more sparse (a greater distance between symbols) there is less potential for overlap. For example, the symbol sums for $\{\overline{3}, 0, 1, 7\} + \{\overline{3}, 0, 1, 7\} = \{\overline{6}, \overline{3}, \overline{2}, 0, 1, 2, 4, 7, 9, 14\}$. Here the sparsity of the operand symbol alphabets means there are no overalps. Hence, the upper limit of (10.5) is reached, with $|\mathbb{S}_\mathrm{I}| = |\{\overline{6}, \overline{3}, \overline{2}, 0, 1, 2, 4, 7, 9, 14\}| = 10$. To accommodate this, more resources are required as explained in Section 5.10.

10.6.2 Conversion cost and performance

Converting between number systems can be costly in terms of both logic resources and performance. If the symbol alphabet being converted to is a super-set of the other alphabet then the conversion is a simple casting from one to the other. This casting is free if the digital encoding is equivalent for the symbols in the intersection of the alphabets.

Conversion from a redundant number system to a non-redundant system requires a many-to-one conversion as explained in Section 6.2. As the symbol alphabet grows there are more redundant representations for each value, making the conversion process necessarily more complicated. In general, a carry propagation the full length of the representation is required, making the performance comparable for various symbol alphabets. However, a symbol alphabet with more or larger symbols will require more hardware to perform the conversion. The hardware required is roughly equal to a redundant adder plus a binary adder. The redundant adder reduces the representation to BSD by adding it to zero (see Section 5.10 for resource usage) and the binary adder converts the BSD representation to binary (see Section 6.2.1 for resource usage).

10.7 SUMMARY

The selection of a number system is application specific and within the application different modules may internally use different number systems. A number system should be chosen that balances the tradeoff between performance and cost. Performance can be gained from various properties of the number system, such as the minimum Hamming weight. The maximisation of relevant properties is highly desirable. With these considerations in mind several number systems come to the fore.

BSD, with a radix of 2 and the symbol alphabet $\{\overline{1}, 0, 1\}$, is simple and balanced, making it easy and efficient to implement. The simplicity is due to the small magnitude symbols, and the dual digital signal encodings for the zero-symbol. Many other representations are reduced to BSD to facilitate computation. However, BSD can only achieve an average minimum Hamming weight of $N/3 + 1/9$, making it less powerful for the partial product packing.

To achieve lower Hamming weights the symbol alphabet needs to be expanded. The symbols chosen should be co-prime to the radix. Expanding to include radix multiples of symbols already in the alphabet does not reduce the Hamming weight. Symbols should have a small magnitude. The larger a symbol is, the greater the logic required to perform an addition. Including at least one negative symbol allows negative numbers to be represented without special treatment. One number system that fulfils these requirements is the $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$, $\beta = 2$ number system.

This and previous chapters have considered integer values for the symbol alphabet and radix. However, there are many other options. The radix may be an irrational number such as the golden ratio, $\varphi = \frac{1-\sqrt{5}}{2} \approx 1.6180339887$ [Bergman, 1957], or a complex number such as $2i$ [Knuth, 1998] or $\pm 1 \pm i$ [Jamil, 2001]. The symbol alphabet might also have fractional symbols. This will be discussed futher as possible future

work in Section 11.4.1.

Chapter 11

CONCLUSIONS AND FUTURE WORK

The work in this thesis has been presented as a number of theoretical ideas on the properties of redundant number systems, with algorithms for realisation of these properties, and utilised in practical implementations for field programmable gate arrays (FPGAs). The specification of the number systems has been kept general, such that a range of number systems can be explored and their properties compared. The ideas have been applied to the design of a dot-product calculator for fast implementation of finite impulse response (FIR) filters.

Redundant number representations are proposed as a path to increasing the performance of digital signal processing (DSP) systems. Through their greater symbol alphabet cardinality they gain exploitable properties not present in non-redundant number systems such as binary. These include lower minimum Hamming weights and the capture of propagating carries in adders. Both of these properties provide opportunities to increase performance in common DSP algorithms. Lowering the Hamming weight reduces the number of accumulations in a multiplication. Adders are constructed that have fast and constant time performance, irrespective of the word lengths of the operands. These two exploits are combined in the implementation of a dot-product unit, an often used functional block in DSP for filtering, correlation, convolution, and beam-forming. Furthermore, an additional exploit was discovered that uses the representation redundancy to further optimise the dot-product, reducing the number of clock cycles required. The property of redundant number systems enabling the dot-product optimisation is identified as zero-dominance. The positions of zero-symbols within a representation is given greater importance than minimising the count of zero symbols. This differentiates the the zero-dominant representations from the purely minimum Hamming weight representations.

A summary of the major contributions presented in this thesis is given in Section 11.1. The situations when a redundant number system can be beneficially employed are discussed in Section 11.2. This is followed by suggestions for future work. Suggested

applications are given in Section 11.3, and suggestions for future theory of redundant number systems are given in Section 11.4.

11.1 SUMMARY OF CONTRIBUTIONS

The research presented in this thesis has developed four major contributions to the area of computer arithmetic that are original, to the best of my knowledge. Each of these major contributions are summarised in the subsections below.

11.1.1 Minimum Hamming weight for irregular alphabets

A method was developed to find the minimum Hamming weight encoded representations in a number system that has an irregular symbol alphabet. The generated finite state machine allows the average and worst case minimum Hamming weights to be calculated. Prior methods, though simple, are specific to a number system or can only recode to a limited selection of number system specifications. This prompted the development of a new method that can also handle *irregular* symbol alphabets.

The new method generates finite state machines that serially recode non-redundant representations into redundant representations with minimum Hamming weight. This is done using only the symbols designated in the alphabet specification. A Markov analysis on the state machine reveals the expected minimum Hamming weight for a random value in the specified number system. Furthermore, a search of the state machine for the elementary cycles with the highest ratio of output nonzero-symbols to the length of the cycle gives the worst case minimum Hamming weight and the representations that achieve it.

A parallel method of minimum Hamming weight recoding a binary word into binary signed digit (BSD) or $\mathbb{S} = \{\overline{1}, 0, 1, 3\}$,$\beta = 2$ was also developed. It performs the recoding in constant time, irrespective of the input word length. The tradeoff for this performance and parallelism is the substantial number of resources consumed.

11.1.2 Zero-dominance redundant representation property

A new property of redundant number representations called zero-dominance is proposed. This property places importance on the location of zero-symbols within a representation. This extends the minimum Hamming weight property where maximising the zero-symbol count is the only criteria.

A new algorithm was developed to find *all* the zero-dominant representations of a value for a given number system specification. This gives the zero-dominant set of representations, which includes all minimum Hamming weight representations as a subset. The zero-dominant set defines a Pareto-frontier that covers all representations of a value such that all possible positions of zero-symbol are represented by a representation with a low Hamming weight. This is used in the optimisation of partial product packing to reduce the number of representations that must be considered.

11.1.3 Efficient implementation of constant time addition in FPGAs

A resource efficient implementation for constant time adders was developed for low-cost FPGAs. The design uses a structural hardware description to utilise the additional logic and routing resources intended for supporting binary ripple-carry addition.

Having representation redundancy at the output of an adder allows for constant time addition. Essential in the implementation is the removal of any signal path that can cause carry propagation. The 3:2 compressor style of addition was shown to eliminate these long signal paths. The 3:2 compressors are the basis of the redundant adder implementation for both the Xilinx Spartan 3 and Altera Cyclone III FPGAs. Two incarnations of the 3:2 compressor are required; one where all input bits have the same sign, and one where one bit has a different sign.

Experiments showed that the redundant adder achieved constant time addition, irrespective of the operand widths on both architectures. The BSD adder implementation outperformed the standard binary ripple-carry adder at word lengths of 44 and 24 positions on the Spartan 3 and Cyclone III respectively. Critical to the success of these results was the structural description of the circuit, utilising the additional fast connections between adjacent logic blocks and its associated logic. This gave a resource efficient implementation that for a BSD adder consumed only two 4-input lookup tables (4-LUTs) and three 4-LUTs per position on the Spartan 3 and Cyclone III respectively. In contrast a behavioural circuit description synthesises to consume four 4-LUTs per position.

This work can also be used for the efficient implementation of Wallace and Dadda trees for binary addition. These structures are realised in the current FPGA architectures, circumventing the requirement for specialist multi-operand addition blocks suggested in other literature.

11.1.4 Dot-product partial product packing and optimisation

A novel packing of the nonzero partial products in a dot-product was proposed to reduce the number of clock cycles for calculation. The number of clock cycles is further reduced by recoding the multiplier operands with minimum Hamming weight representation. The packing is maximised by an innovative use of redundancy in representation of the coefficients. An algorithm was developed to optimally choose representations from the zero-dominant sets of the coefficients.

The multiplicative absorbing property of zero is used to eliminate partial products in the multiplications of a dot-product. The remaining significant (nonzero) partial products are packed together so that partial products from different multiplications and with different fixed shifts are mixed together. This forms a mixed parallel multiplication where there is no time or energy wasted accumulating zero valued partial products. The resulting out-of-order accumulation of the dot-products is valid since fixed point addition is associative. A redundant number system was applied to the multiplier operands, initially to take advantage of lower Hamming weight; reducing further the number of partial products.

A novel opportunity was identified to further improve the packing by choosing redundant representations for the multiplier operands that mesh together well. This required the development of a combinatorial optimisation algorithm to choose one representation for each multiplier operand. The target is to minimise the number of partial products with the same shift, simultaneously over all shifts. The zero-dominant set facilitates the optimisation algorithm by providing a Pareto-frontier from which to select representations.

Currently the actual redundancy in representations is rarely, if ever, used in the design of algorithms. The approach taken for optimising the packing shows that redundant number systems have uses beyond the minimum Hamming weight property. Another application area where the full redundancy can be used in a novel way is encryption as suggested in Section 11.3.2.

11.2 WHEN A REDUNDANT NUMBER SYSTEM SHOULD BE EMPLOYED

Considering an application system as a whole and which *single* number system would be best to use throughout, the conclusion would likely be in favour of two's complement binary. It best matches the current digital technology. This is the choice made for general purpose digital signal processors, where the processing elements are designed

with little knowledge of the final algorithms to be run. However, with an FPGA, the designer is not restricted to a single number system, and can consider subsystems individually to choose an appropriate number system to use at each stage.

Redundant number systems can give a performance improvement over a standard binary implementation. Often it is only a small part of a DSP system that requires high performance. This part usually sets the minimum performance criteria of the device, despite the majority of the system having a lower performance criteria. For instance, the high frequency front end of a software radio requires a high performance band-select filter, after which it is converted to a lower intermediate frequency and/or baseband for demodulation and bulk processing. To implement the band-select filter in binary may require a high speed and therefore, high cost FPGA. A lower performance and hence, lower cost FPGA may suffice for the remaining processing. A redundant number system could be used to enhance the performance of the high performance subsystem, for example, the band-select filter, allowing it to meet its performance criteria in a lower speed grade FPGA. While the redundant number system implementation may consume more resources than a binary system, it is only a small part of a larger system. Thus considerable cost could be saved as the entire system can be implemented in a cheaper FPGA with performance better suited to the majority of the processing chain.

However, taking this partitioning idea to the extreme and breaking the DSP chain into fine grained parts, single adders for instance, does not promote the use of redundant number systems. The performance is then dominated by the conversion between number systems. This can cause a considerable delay, such that it negates the use of redundant representations. The performance penalty of this conversion must be amortised over a number of fast redundant number operations before a benefit can be realised. For example, the dot-product design amortises the final conversion to binary over hundreds of additions with intermediate or fed-back results kept in the redundant format.

11.3 FUTURE WORK - APPLICATIONS

This section suggests some future application areas where the utilisation of redundancy in the number representations may yield benefits. These benefits may be speed improvements, reduction in energy consumption, or enhanced functionality.

11.3.1 High performance and high precision IIR filters

Limited word lengths can cause several detrimental effects in infinite impulse response (IIR) filters. These include changes in frequency response due to the quantisation of

coefficients, overflow due to the feedback path, and round off noise or limit cycles from internal rounding of intermediate results in the feedback loop. The small word lengths that cause these effects are used because the speed of the arithmetic at larger word lengths is too slow. Employing constant time adders within the filter decouples the performance from the word length, allowing longer word lengths to be used internally, minimising the effects. Only at the external interface is the result reduced in word length by saturation and rounding. Small word length results can be obtained with the stability and accuracy of a large word length implementation.

11.3.2 Hardening encryption algorithms against side-channel attacks

Redundant number systems may be effective at hardening cryptographic algorithms to side-channel attacks. This is another area where the redundancy of the number system is used, rather than just the minimum Hamming weight property and the constant time addition.

Side-channel attacks are a form of cryptanalysis where the attacker can gleam additional information from measurements made of the crypto-system. These may include making detailed timing, power usage, or electromagnetic radiation measurements. From this information the attacker can correlate against stochastic models of the crypto-system to infer the cryptographic key or internal secret states [Standaert, 2009]. To expedite the process the attacker may provide texts to be encrypted that have particular properties such as a known Hamming weight. Critically, these attacks are aimed at the implementation of an algorithm. Where the algorithm itself may be secure to analytic or brute force attack taking many centuries to guess the correct key, a poor implementation subjected to side-channel attack can be broken in minutes.

Side-channel attack techniques depend on the correlation between the secret key and the execution flow of the algorithm. Hardening of the implementation requires the designer to de-correlate the secret key dependence by either removing it completely and giving every execution the same side-channel signature, or randomising the signatures, for example adding random delays or operations.

A countermeasure may be to use redundant number representations in the implementation to randomise the execution. Having a redundant number system for the secret-key and computations would allow the key's representation to be mutated without affecting its value, hence, changing the implementation but not the algorithm. Different representations for the key would produce alternative side-channel signatures as the execution flow is altered and components like adders execute differently. Periodically choosing a random key representation would change the side-channel signature between

subsequent executions on the same cipher-text. Furthermore, selecting random representations for the internal cipher-text would additionally obscure the side-channel profile. Since the Hamming weight can change between representations, attacks predicated on inserting data with a known Hamming weight are disrupted. From the perspective of a side-channel attack the effective key length of the implementation is increased while maintaining the external interfaces and the key length of the other party.

11.3.3 Energy requirements of constant time adders in FPGA

An important consideration for mobile electronics is energy consumption. The energy consumed by a device is affected by many variables including the voltage, the frequency of operation, the number of transistors, and how often the transistors switch. The more complex redundant adders presented in Chapter 5 all necessarily require more silicon area (number of transistors) than a binary ripple-carry adder. This serves to increase the dynamic energy required for an addition. However, in a ripple-carry adder there is substantial switching as the carry propagates along the word changing the state of the carry and sum signals multiple times. This is called glitching and results as different paths though the combinational logic take different times to converge to a steady state. Since there is no carry-propagation in redundant adders, the glitching should be reduced, resulting in a decrease in energy consumed. As indicated by Smitha et al. [2006], the reduced switching from less carry propagation in a redundant adder outweighs the increase in transistor count, resulting in less energy consumed for the same performance. This study was performed for ASIC implementations of hybrid signed digit adders of Phatak and Koren [1994] and their own improved implementation. Can this reduction in energy consumption per addition be replicated in FPGAs?

To achieve the best results, it may require effort to manually place and route the adder designs to achieve a regular layout with the aim of maximising performance and minimising the capacitive loading on the logic.

11.4 FUTURE WORK - THEORY

This section suggests some further possible areas of investigation in the theory of redundant number representations.

11.4.1 Zero-dominant set maximisation and fractional symbols

The zero-dominance property of redundant representations singles out some representations as having a particular placement of zero-symbols. Since these representations have proven to be useful in the PPP optimisation, the maximisation of the zero-dominant set cardinality may provide better or easier packing. This requires finding symbol alphabets that enhance the zero-dominance property.

Symbols alphabets with fractional values may be one way of engineering different nonzero-symbol placements. For example, the value three in binary is represented as 011_2, which can be recoded as $10\overline{1}_2$ or 003_2. All these representations use a symbol in the least significant position. A fractional symbol of $\frac{3}{2}$ can give the value three as $0\frac{3}{2}0_2$ without a symbol in the least significant position, hence increasing the zero-dominant set for value three. In general, a fractional symbol should have a denominator that is a power of the radix. This would give a symbol equivalent to the numerator but left shifted by a number of positions equal to the power.

Adders would require carry symbols that move right. For example, a $\frac{3}{2}$-symbol would be recoded to a sum symbol of 1 in the same position and a fractional carry of 1 to the position below. Sign detection would also be facilitated as all the symbols can have a magnitude less than the radix. Thus the most significant nonzero symbol will accurately determine the sign.

11.4.2 Improvements to partial product packing

The process for building a FIR filter in Chapter 8 began with the design of the filter coefficients. These were fixed and not changed. If the PPP and optimisation process is allowed to influence the design of the filter coefficients, a further improvement in throughput may be achieved. It has been demonstrated by Samueli [1989] that a satisfactory filter design can be achieved with restrictions on the coefficients' legal values. In this case, only coefficients with a total weight of one or two in BSD are permitted. This introduces errors into the filter beyond that of quantisation error and to compensate the filter is extended in length.

The idea is to amalgamate the design of the FIR filter coefficient values with the optimisation and partial product packing in a redundant number system. This may initially entail avoiding representations with worst case minimum Hamming weight, such as in [Samueli, 1989]. However, as identified in the PPP, the positioning of zero-symbols is also important to achieve a good packing. Ideally, the coefficient values would be modified to improve the packing by giving zero-dominant representations that

reduce a long row. This may be achievable without introducing too great an error and consequently having to extend the filter length too much to compensate.

11.4.3 Minimum Hamming weight distributions

In this thesis the average and worst case minimum Hamming weight factors were calculated from a recoding state machine. Value zero has of course the best case Hamming weight of zero. The distribution of minimum Hamming weights derived from all values may provide some additional information useful in selecting a number systems. Fitting the histograms to a minimum hamming weights to a parametrised distribution would facilitate a probabilistic analysis of the PPP.

Appendix A

BINARY ADDITION STRUCTURES TO IMPROVE CARRY PROPAGATION

In order to improve the speed of binary addition many extensions have been proposed by J. Sklansky [1960], Brent and Kung [1982], Kogge and Stone [1973], Huey Ling [1981], Kilburn et al. [1959], O. J. Bedrij [1962], Doran [1988]. These improvements focus on speeding up the carry propagation, although by doing so they increase the size of the logic required. Therefore these improved binary adders trade an increase in resources (cost) for decreased delay (increased performance).

A.1 CARRY LOOK-AHEAD ADDER

The carry look-ahead adder removes the carry chain and replaces it with parallel carry generation logic, referred to as carry look-ahead logic [A. Weinberger and J. L. Smith, 1958, John F. Wakerly, 2000, Hwang, 1979]. This means that the half adder elements at the end do not need to wait for their carry input to propagate. The carry C_i into position i is calculated as a function of the input bits of operands A and B up to position i; $(A_i, A_{i-1}, \ldots, A_0, B_i, B_{i-1}, \ldots, B_0)$ and the initial carry into position zero (C_0). Two terms are used for describing the creation of a carry and stem from the two conditions that cause a carry in a ripple-carry adder.

1. A *carry generate* occurs when both A_i and B_i are 1. Their sum is two without a carry in or three with a carry in. Either way a carry into the next position is required. Therefore, the carry look-ahead logic will always generate a carry in this case regardless of the carry into that position. Hence the boolean equation for a *carry generate* is,
$$G_i = A_i \bullet B_i. \tag{A.1}$$

2. A *carry propagate* occurs when one of A_i or B_i is 1. Their sum is one without a carry in and two with a carry in. Therefore, a carry out will only be propagated if

there is a carry into that position. That is to say that the carry in will propagate to the carry out. The carry look-ahead logic will propagate a carry in this case. Hence the boolean equation for a *carry propagate* is,

$$P_i = A_i \oplus B_i. \tag{A.2}$$

A carry into the next position (C_{i+1}) will occur if there is a *carry generate* condition or if there is a *carry propagate* condition and previous carry (C_i) is a '1',

$$C_{i+1} = G_i + P_i \bullet C_i. \tag{A.3}$$

If the carry look-ahead logic was implemented simply using equation A.3, then a carry chain would still exist because it depends on the previous carry. However equation A.3 can be written as a recursive relationship depending on the previous carry C_i, which in turn can be found with G_{i-1}, P_{i-1} and C_{i-1}. The recursion terminates when it reaches C_0. The recursion is expanded in equation A.5 to a sum of products for the first four carries.

$$\begin{aligned} C_1 &= G_0 + P_0C_0 \\ C_2 &= G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0 \\ C_3 &= G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\ C_4 &= G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned} \tag{A.4}$$

Since the carry-lookahead logic can be expressed as a large sum of products, all carries are calculated in parallel. This means an addition of arbitrary bit width should be calculated in constant time. However this does not take into consideration the practical implications of the implementation technology. In a full custom logic design the number of terms in each equation is limited by the number of inputs to each gate (fan-in) and the number of gates that can be driven by a particular output (fan-out). In an FPGA the number of inputs is limited to the size of the look-up table (typically four-input) and the fan-out (typically 32 inputs). When these limits are reached another layer of logic is required, increasing the propagation delay of the circuit. This is detailed in section C.1.

A.2 PREFIX TREE ADDER

Prefix tree adders are derived from carry look-ahead adders and attempt to solve the various problems of fan in, fan out and resource usage[Brent and Kung, 1982, Kogge and Stone, 1973].The fan-in and fan-out limitations can be minimised by employing a

divide and conquer approach to the carry look-ahead logic.

1. The carry look-ahead generate and propagate signals are calculated for each input pair using equation A.1 and equation A.2 respectively.
2. The pairs of generate and propagate signals $\{G_0, G_1, P_0, P_1\}$ and the carry in signal (C) are used to create carry groups with the output signals,

$$\begin{aligned} G &= G_{i+1} + P_{i+1} \cdot G_i && \text{(A.5)} \\ P &= P_{i+1} + P_i && \text{(A.6)} \\ C_0 &= C && \text{(A.7)} \\ C_1 &= G_0 + P_0 C. && \text{(A.8)} \end{aligned}$$

3. The group generate and propagate signals are paired again by another group.
4. The carry out signals are passed down through the groups below and finally to the adders to generate the sum,

$$S = (A \oplus B) \oplus C. \quad \text{(A.9)}$$

This builds a tree of two input carry look-ahead adders. The structure for a 8-bit adder is shown in figure A.1 and is known as the Brent-Kung adder[Brent and Kung, 1982]. To make a 16-bit adder figure A.1 would be duplicated and joined with a fourth level of group carry logic.

The original carry generate and propagate equations from equation A.5 are split into sub-expressions and shared amongst the carries. This reuse of subexpressions reduces the total amount of logic required, however the total propagation delay is increased. A critical path is also shown in figure A.1 traversing the tree up the least significant side through the carry propagate signals and down the most significant side through the carry signals.

A.3 CONDITIONAL SUM ADDER

Conditional sum adders calculate two sums and two carries for each operand bit pair by assuming a carry in of '1' and '0' respectively. The tree structure shown in figure A.2 then selects the correct sum and carry based on the sum and carry already resolved in lower positions [J. Sklansky, 1960].

The performance is increased through precomputing all possible results and quickly selecting the correct one with multiplexers as the carry becomes available. This gives

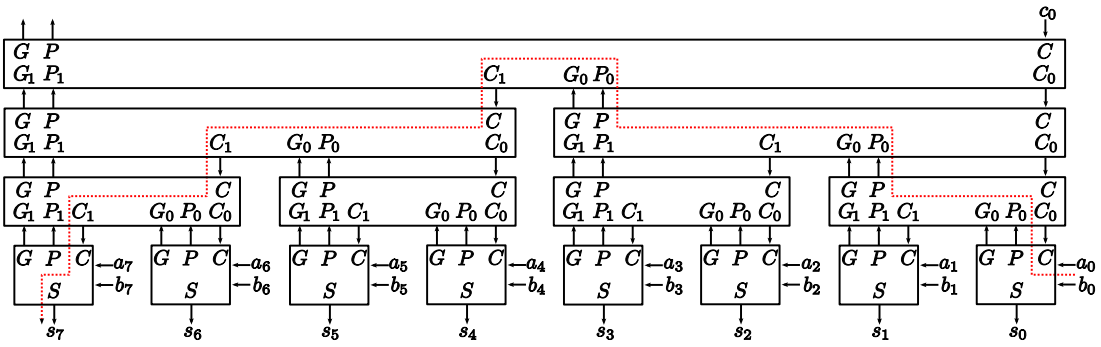


Figure A.1: Sixteen bit prefix tree adder using the Brent-Kung structure from [Brent and Kung, 1982]. The critical delay-path is shown by the dashed line.

the conditional sum adder a $O(\log N)$ perfromance. The resource usage however also grows quickly.

A.4 OTHER ADDER STRUCTURES

There are many more adder structures that attempt to solve different problems. Some structures are hybrids of other structures that provide a different balance between resource usage and performance. These structures include,

1. **Ling Adder**. An improved version of the conventional carry look-ahead adder proposed by Huey Ling [1981] and generalised by Doran [1988]. It is faster and uses less resources though modifying the carry generate and propagate functions so that they do not use XOR gates.

2. **Carry Skip Adder**. Modifies the carry chain to quickly skip full adders if the carry propagate signal is high [Kilburn et al., 1959]. Uses ideas from both the carry-look-ahead and carry-select adders. It still has a carry chain so has a delay of $O(N)$.

3. **Carry Select Adder**. This structure is a hybrid of the ripple-carry adder and the conditional-sum adder. The adder is broken into blocks of k bits. Each addition block is implemented twice as short ripple carry adders, once assuming no carry in and once assuming a carry in of one. The result from each block is selected using a multiplexer to conditioned on the carry out of the previous block [O. J. Bedrij, 1962]. This adder is also $O(N)$, however faster than ripple carry taking a time of $k + N/k$ full adder delays.

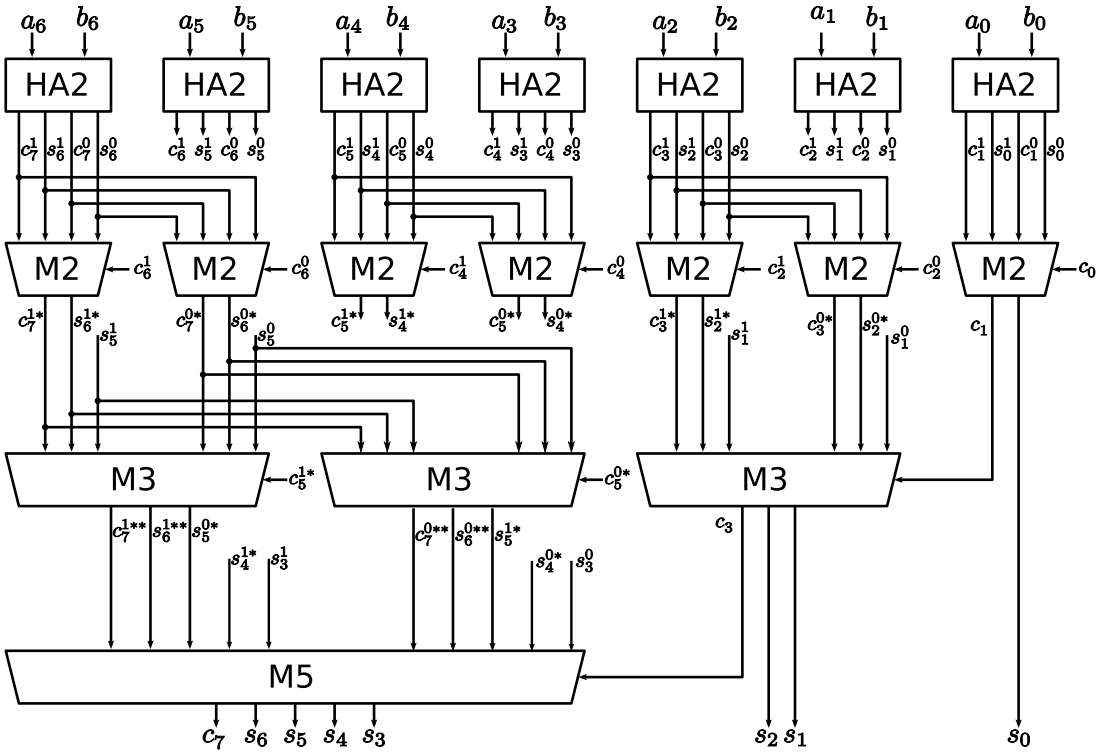


Figure A.2: A seven bit conditional sum adder structure. HA2 is two half adders, M# is a multiplexer that selects # inputs from the left if the select line is high, otherwise it selects the inputs on the right.

Appendix B

FIELD PROGRAMABLE GATE ARRAY

This appendix provides an overview of the low-cost field programmable gate arrays (FPGAs) used in this thesis. In particular the architectures of the Altera Cyclone and the Xilinx Spartan Families of FPGA.

B.1 ALTERA CYCLONE III ARCHITECTURE

The architecture of the Altera Cyclone series of FPGA is relatively simple. The structures are regular without any unnecessary complication. The basic logic block is called a logic element. These are grouped into sets of 10 to form a logic array block (LAB) with internal local routing and shared control signals and two clock signals. LABs are arranged in columns and are interconnected by the global routing network.

B.1.1 Logic elements

The logic block of a Cyclone series FPGA is called a logic element (LE). It consists of a four input look-up table (4-LUT) and a D-type flip flop. The LE can be configured in one of two modes, the normal mode for regular logic description, or arithmetic mode that activates special routing for performing binary arithmetic. A schematic of an Altera Cyclone III logic element in normal mode is shown in Figure B.1.

The combinational logic function is solely provided by the 4-LUT, capable of implementing any four-to-one logic function. The 4-LUT is a 16 by 1-bit random access memory (RAM) with the four inputs decoded as an address into the RAM. The combinational function output is stored in the RAM at the address encoded by the inputs. Changing the inputs selects the appropriate RAM contents to the LUT output. This is shown in Figure B.2 where the input decoding is performed by a binary tree of multiplexers.

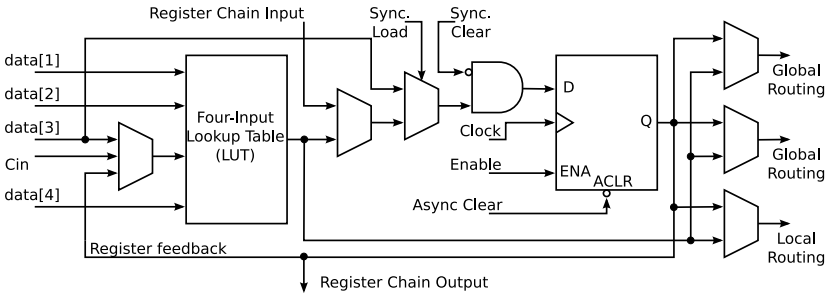


Figure B.1: The logic diagram of an Altera Cyclone III logic element in 'normal mode' [Altera Cyclone III, 2008].

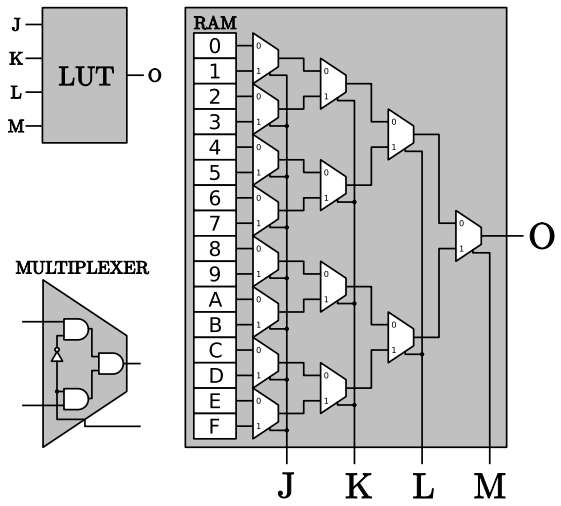


Figure B.2: Simplified logic diagrams for a 4-input {J,K,L,M}, single output {O} LUT in normal mode showing the internal address decoding of the RAM table using multiplexers.

A D-type flip-flop provides a single bit of memory to register the 4-LUT output. The register is clocked on the rising clock edge and has an enable line. Additional control lines provide synchronous loading and both synchronous and asynchronous clearing of the flip-flop. The flip-flop is commonly referred to as a register.

The logic elements can be used in two modes. Appropriate use of the logic element modes allows the efficient implementation and optimisation of logic designs. The modes mostly influence the input and outputs of the 4-LUT.

1. Normal mode uses the LUT to implement up to any four-input to one output combinational logic function as shown in Figure B.2.

2. Arithmetic mode allows the LUT to simultaneously implement two combinational functions that share three inputs. Two inputs are from the general interconnect routing and the third is the carry-in signal from the previous LE. The LUT generates two outputs instead of one. The first output is routed to the register or general interconnect as usual and the second is routed to the next LE's carry-in signal.

Arithmetic mode of a logic element, shown in Figure B.3, is commonly used to efficiently implement full adder circuits that calculate both a sum output and a carry-output from two input bits and a carry-in. Without arithmetic mode, two logic elements would be required to calculate the sum and carry-out independently. To provide this functionality, the 4-bit look-up table is split into two 3-input look-up tables (3-LUT) sharing common inputs as shown in Figure B.3 and Figure B.4 [Cliff and Cope, 1993]. One 3-LUT to calculate the carry-out and the other for the sum output. The carry-out bit is only routed to the next neighbouring logic element in the carry chain and the sum bit is routed to the normal LUT output. Combining both the carry-out and sum combinational functions in a single logic element in arithmetic mode halves the logic usage and the use of short dedicated routing for the carry bit drastically decreases the routing delay [Altera Cyclone III, 2008].

Normally the flip-flop would register the output of its associated 4-LUT, however, additional register configurations allow the register and LUT to be used independently. These register configurations and their function are:

1. Register bypass, connects the LUT output directly to the external routing. This allows combinational functions of greater than four inputs to be constructed as discussed later in section C.1. This leaves the register unused and available for functions requiring additional sequential logic or memory.

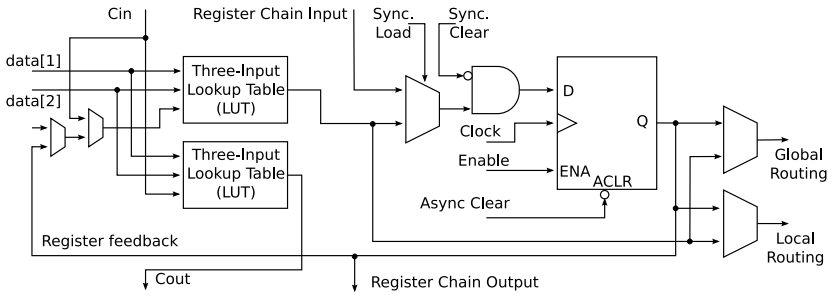


Figure B.3: The logic diagram of an Altera Cyclone II logic element in 'arithmetic mode' [Altera Cyclone III, 2008].

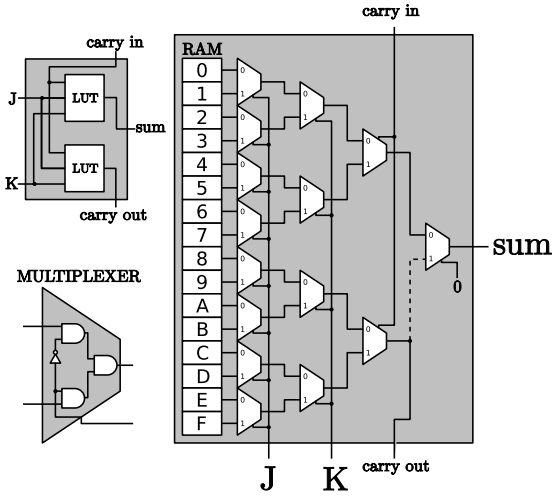


Figure B.4: Simplified logic diagrams for a LUT in arithmetic mode where it is split into two 3-input LUTs to provide both a carry-out and sum outputs.

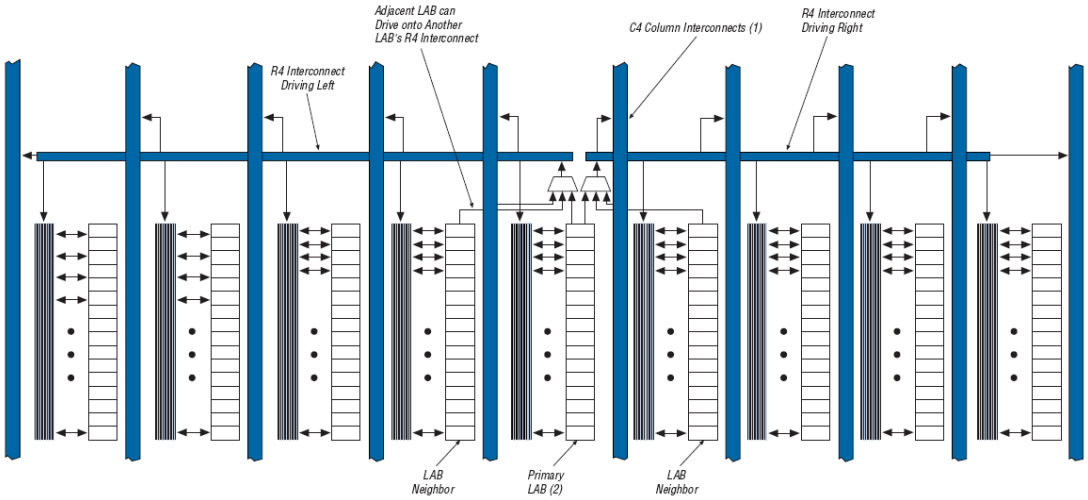


Figure B.5: Layout of the Cyclone III showing logic elements, LABs in columns and interconnects [Altera Cyclone III, 2008].

2. Register chaining, takes the output of the neighbouring register as the input to the register, bypassing the LUT. This facilitates the compact construction of shift registers and fast random access memory.

3. Register packing, provides a similar function to register chaining, however, the register input may be from any signal, typically another register such as in a pipeline delay stage. The input is accessed from the DATA3 input of the LUT which may limit the LUT to three inputs if that particular signal is not used in the LUT function.

B.1.2 Hierarchical structure

There is a hierarchical structure to the grouping of logic elements and the routing between them as shown in Figure B.5. The logic elements are grouped into logic array blocks (LABs) and the LABs are grouped into columns. Each FPGA device has multiple columns. Each LAB contains 10 logic elements, which share the control signals: clock, enable, synchronous load and clear and asynchronous clear. There is local routing between the outputs and inputs of logic elements within the LAB. The local routing has a lesser delay than the global routing. As previously discussed, special connections exist between neighbouring logic elements for register chaining and the fast carry chain. These special routes also cross the LAB boundaries within columns but not between columns.

B.2 XILINX SPARTAN 3 ARCHITECTURE

The architecture of the Xilinx Spartan series of FPGA is similar but more complicated than the Altera's Cyclone. It includes additional gates and multiplexers in the basic logic blocks. These help to reduce the number of look-up tables used to implement some common logic structures such as wide multiplexers or binary multipliers.

B.2.1 Slices

The basic unit in the Spartan 3 architecture is the slice. A slice contains two logic blocks, each providing at least the equivalent functionality of a Cyclone logic element in normal mode. Figure B.6 shows a logic schematic of the Spartan 3 slice. It has two 4-LUTs per slice and two flip-flops. Two flavours of slice exist; a SLICEL that provides this basic functionality, and a SLICEM that can alternatively provide additional distributed memory and shift register functionality using the LUT RAM.

Resource efficient ripple carry arithmetic is also supported in the slice but with a different approach than the Cyclone. A carry chain extends up a column between adjacent slices. The carry chain uses a carry look-ahead style logic for each bit to speed up carry propagation as the carry needs only pass through a 2:1 multiplexer [Xilinx Spartan-3, 2008]. This allows a propagating carry signal to quickly pass full adders where the propagate signal is high. See appendix A.1 for an explanation of carry look ahead adders.

The logic of Figure B.7 is used to construct a full adder. The 4-LUT (F) in Figure B.6 is used to generate the propagate signal $P = A \oplus B$. The multiplexer CYMUXF selects the carry-out signal as,

$$C_{\text{out}} = PC_{\text{in}} + \overline{P}G, \tag{B.1}$$

where G is the generate signal selected by the static multiplexer CYOF as either A or B, $(G = A)$. This is different to the usual generate function of $G' = AB$ but equivalent,

$$\begin{aligned} G' &= \overline{P}G \\ &= \overline{(A \oplus B)}A \\ &= \overline{A}A\overline{B} + AAB \\ &= AB \qquad \square. \end{aligned} \tag{B.2}$$

Finally, the sum signal is generated as $S = P \oplus C_{\text{in}}$ using the dedicated exclusive-or gate XORF.

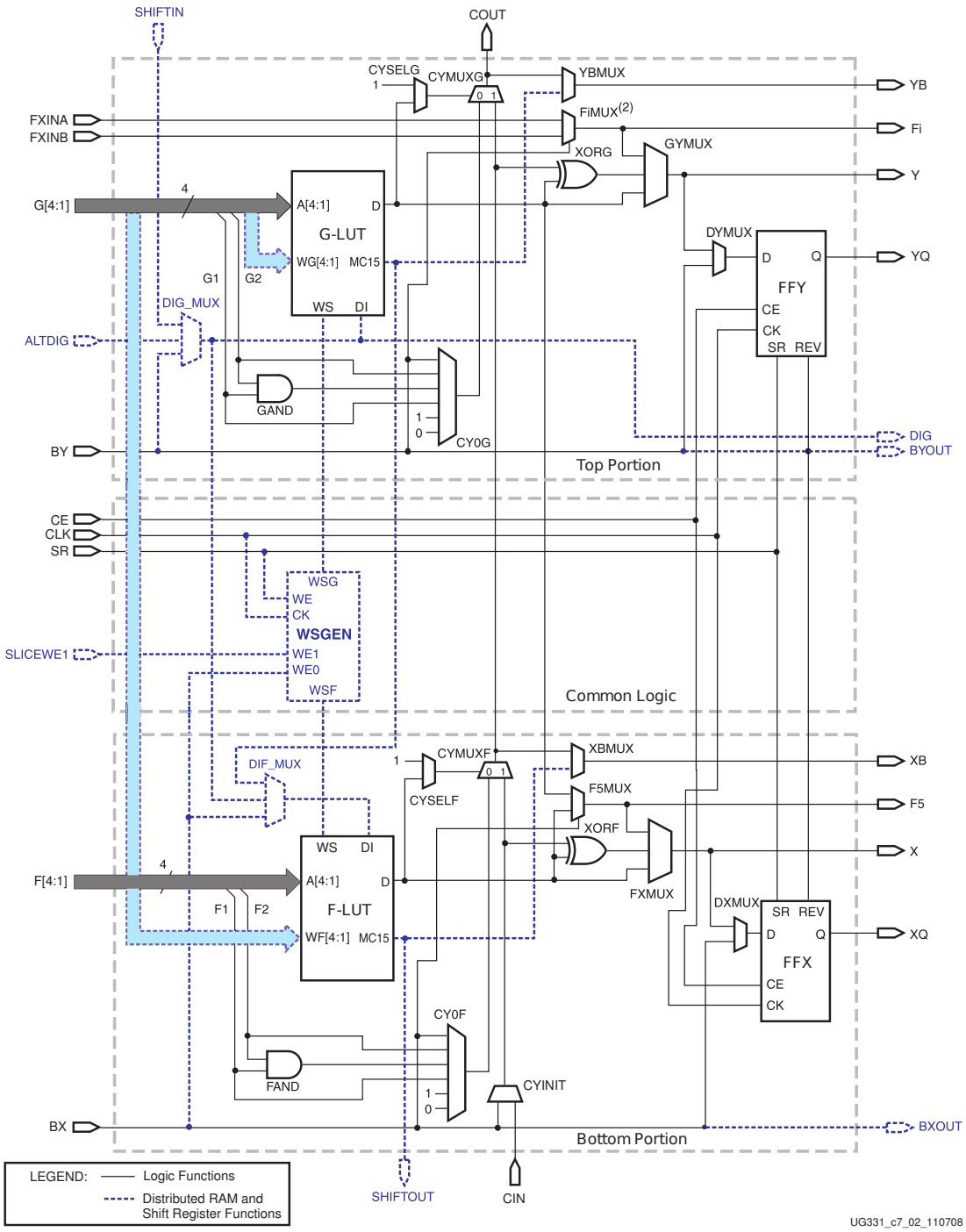


Figure B.6: A simplified logic diagram of the Spartan-3 slice [Xilinx Spartan-3, 2008]. This is the left hand slice (SLICEM) of the complex logic block that includes additional memory functions. The right hand slice (SLICEL) does not include these memory function marked with the dotted lines.

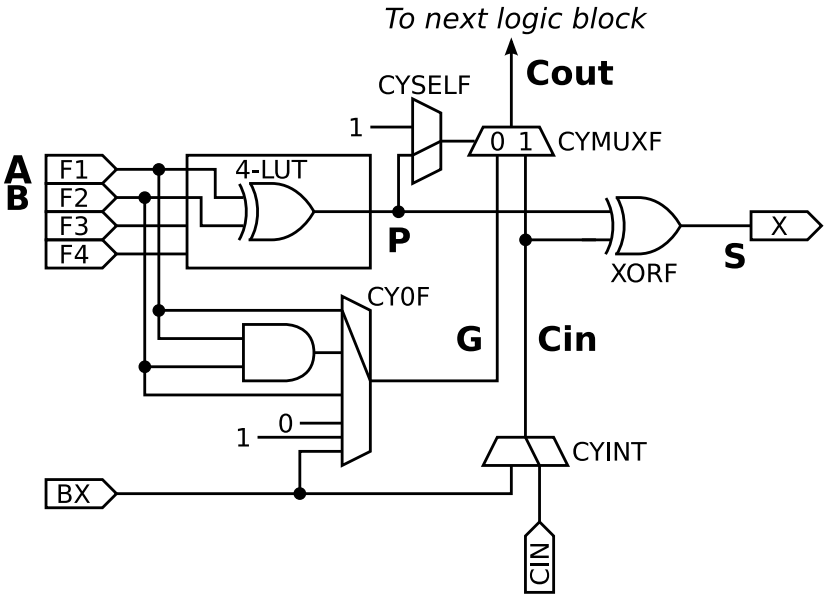


Figure B.7: A simplified logic diagram of the lower half of a Spartan-3 slice showing the logic used to implement a binary full adder.

The carry signal can be initiated through the BX input by changing the static multiplexer CYINT at programming time. The carry can also be terminated into the general interconnect through the static multiplexer XBMUX.

B.2.2 Hierarchical structure

A Spartan-3 FPGA is partitioned into several different levels as shown in Figure B.8. At the lowest level are the slices containing two LUTs and two registers. Four slices are grouped into a configurable logic block (CLB), analogous to an Cyclone LAB. The CLBs are arranged in columns in the FPGA, with each FPGA having multiple columns. Two arithmetic carry chains per column ascend through the CLBs, each chain using two slices. For larger devices both the length of the columns and the number of columns increases.

B.3 DIFFERENCE BETWEEN SPARTAN 3 AND CYCLONE III FPGAS

For all their similarities the Altera Cyclone and Xilinx Spartan FPGAs have been developed from different corners of the size-performance tradeoff. With the Spartan 3 architecture, the Xilinx engineers have targeted improved logic density. That is a greater portion of user logic implemented per slice. This is achieved by providing the

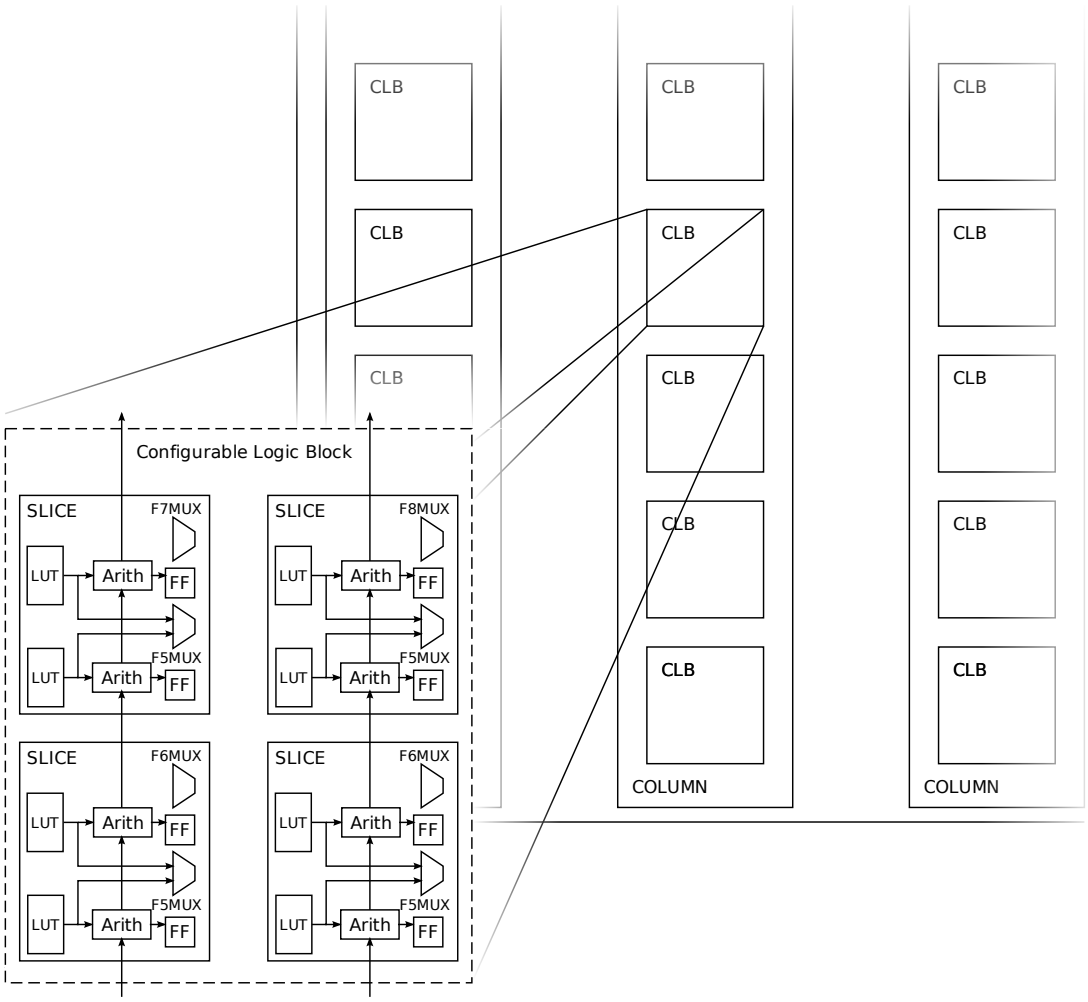


Figure B.8: Spartan 3 heirachy showing the how the FPGA is divided into columns of configurable logic blocks that each contain four slices and two carry chains.

additional F*i*MUX multiplexers that allow synthesis of larger combinational functions using fewer look-up tables, see appendix C. They also provide other logic with the same goal such as the MULTAND AND gate for creating two partial products and their full adder in the same logic block.

The Altera engineers have attacked the performance side of the tradeoff. The Cyclone III logic elements are less complex and subsequently less capable. However, this also means there is less fanout for each logic output and hence lower capacitance for the gates to drive. This means that signals can switch faster, reducing the critical path delay. This speed difference is notable in the performance of the adder implementations presented in sections 5.8.4 and 5.9.2.

B.4 SPECIALISED LOGIC BLOCKS

There are several specialised logic blocks included in modern FPGAs that provide common functions with faster and more compact implementations than possible with the logic elements. The specialised blocks include,

1. 18-bit multipliers. These provide both signed and unsigned binary multiplication with full output width of 36-bits. In the Cyclone III these may also be partitioned into two 9-bit multipliers. To make larger than 18-bit multipliers, the multipliers must be cascaded. For example, to create a multiplier between 19 to 36-bits requires four 18-bit multipliers arranged in a square.

2. Dual port RAM. These provide true dual port RAM, where the ports read and write the same contents but are otherwise completely independent. Each port has it's own address and data signals, clock and enable signals. In the Cyclone III these RAM blocks have up to 9 Kbits of storage each when configured as a 9-bit wide by 1024 deep RAM. It can also be configured with data widths of between 1 and 36 bits. Similarly, the Spartan 3 has RAM blocks of 18 Kbits storage each. The data width is also configurable as powers of two or powers of nine between 1 and 36. The address space is similarly sized such that it fits into 18 Kbits; some bits maybe unused.

3. Phase locked loops. These provide clock multiplication, division, and skew adjustment.

The RAM and multiplier blocks fit into dedicated columns in the FPGA architecture, often next to each other with additional routing between the RAM and multipliers to quickly transfer operands.

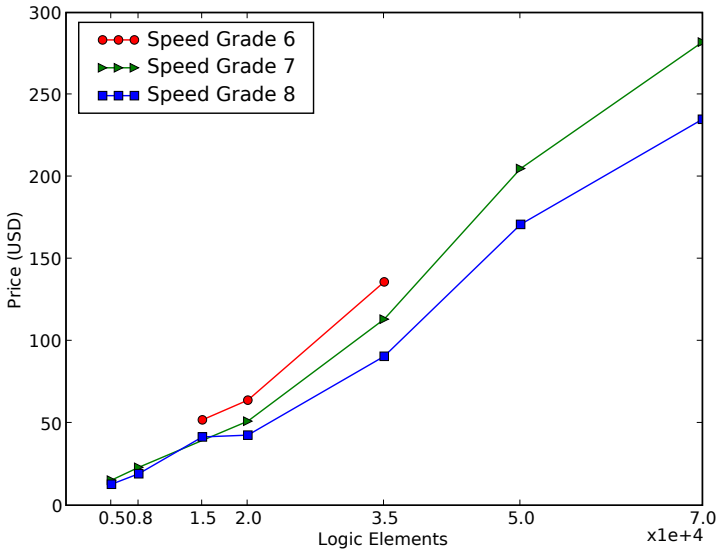


Figure B.9: The cost of single Altera Cyclone II FPGA in March 2007 from the Altera eStore [Altera Corporation, 2007]. Graph showing the combinations of size in logic elements and speed grades and their price.

B.5 PRICE OF FPGA RESOURCES AND PERFORMANCE

The cost of FPGA resources and performance are set by the manufacturing process. The main difference between the members in a family of FPGA such as the Cyclone and Spartan families is the technology of the manufacturing process. As manufacturing technology improves the feature sizes decrease. This increases the total number of logic blocks in a unit area of silicon. It also decreases the capacitive loading on the signal line; increasing the switching speed. Therefore, as the manufacturing technology improves a similar device to the previous technology is smaller and faster. This is reflected as a decrease in the price for the new device.

The devices are categorised in terms of sizes and speed grades. The price of an FPGA increases with both increased size and higher maximum clock rates. This is shown for the Altera Cyclone II FPGA Family in Figure B.9. The speed grades indicate relative performance between two devices of the same size. The maximum frequencies that these devices may run at are shown in Table B.1.

For the same price increase, extra resources could be purchased instead of additional speed. The additional increase in logic is shown in Table B.2 for the C8 speed grade.

Table B.1: Altera Cyclone II speed grades and their maximum performance. Percentage increases are relative to speed grade C8 and prices are based on a 35,000 logic element device.

Speed Grade	Max Frequency	Performance Increase	Price	Price Increase
C8	340MHz	0%	$ 90.70	0%
C7	380MHz	11.7%	$ 113.30	24.9%
C6	420MHz	23.5%	$ 136.00	49.9%

Table B.2: Altera Cyclone II speed grades and their maximum performance. Percentage increases are relative to speed grade C8 and prices are based on a 35,000 logic element device.

Price	Price Increase	Resources (Logic Elements)	Resource Increase
90.70	0%	35,000	0%
113.30	24.9%	39,200	12.0%
136.00	49.9%	43,400	24.0%

Appendix C

IMPLEMENTING WIDE INPUT FUNCTIONS

The Altera Cyclone III and Xilinx Spartan 3 FPGA architectures enforce the use of four input functions, see appendix B. Four input LUTs are provided because they give the best area-delay product [Amit Verma, 2007], hence, giving the best balance between speed and area. To create combinational functions of wider than four inputs, the LUTs are cascaded to create a cone of logic that reduces all inputs to a single output.

C.1 GENERAL METHODS

There are several ways of arranging the LUTs to perform a wide input function,

1. Sum of Products (SOP) / Product of Sums (POS). The inputs are combined using a tree structure of four input look-up tables. The top level LUTs reduce four of the inputs to an intermediate result. Levels of LUTs below combine intermediate results and any remaining inputs until a final result is produced as shown in Figure C.1. This method can achieve the lower limit on the number of LUTs as,

$$\text{LUTs} = \lfloor (I+1)/3 \rfloor, \qquad I > 0. \tag{C.1}$$

   where I is the number of inputs. This limit is achieved when the inputs can be appropriately factored into groups of four to give intermediate results that can also be factored into groups of four. A simple example is an equality comparison between two data buses.

2. Multiplexer. This method builds a binary tree of multiplexers using LUTs as shown in Figure C.2. Each multiplexing LUT has two branches with different results that depend on a further input that decides which branch to select. The LUT in each branch is programmed knowing the path selected by later 'multiplexing' inputs. This can also be thought of as creating a larger look-up RAM by using the additional inputs to create a larger decoder, extending that in Figure B.2.

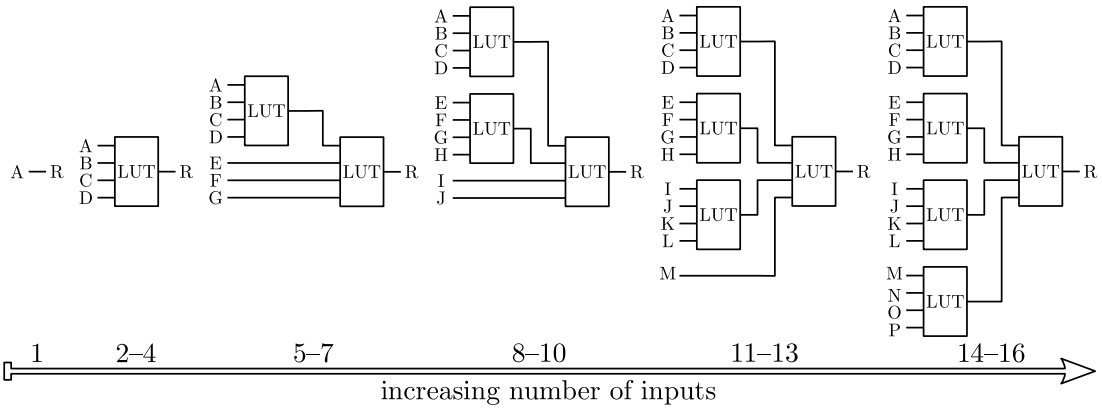


Figure C.1: Diagram showing how the cone of LUTs/logic grows linearly with the number of inputs when using the SOP method.

This method can implement any logic function and gives the upper limit for the number of LUTs as,

$$\text{LUTs} = 2^{I-3} - 1, \qquad I > 4, \tag{C.2}$$

To reach this limit the logic function must be unfactorisable. Examples of functions requiring this method are a multiplexer and an "only x out of y" circuit.

The method chosen is dependent on the desired target of either speed or area and the complexity of the Boolean logic equation to be implemented. When creating a cone of logic, by increasing the number of LUT levels, the propagation delay from the inputs to the output is also increased. Practically, the best solution is to constructed a hybrid of both methods. Either way, the number of logic elements increases drastically with additional inputs as shown by the shaded area in Figure C.3 bounded by (C.1) and (C.2). At best the number of LUTs increases linearly with the number of inputs and at worst it increases exponentially. The use of input functions greater than four is highly undesirable and should be avoided if possible.

C.2 SPARTAN 3 MULTIPLEXERS

The Xilinx Spartan 3 architecture accommodates wide input logic functions using fewer 4-LUTs than the Cyclone. Additional multiplexers are supplied within the slices to combine the outputs of two 4-LUTs. These are collectively referred to as F*i*MUX, where i is a number between 5 and 8 inclusive, indicating the input width. As shown in Figure B.6, each slice contains a F5MUX that selects the output of the F-LUT or the G-LUT based on the 5th input BX. The output signal F5 is either routed to the general interconnect or to the input of a F6MUX, with the selecting signal BY. Two F6MUX feed a F7MUX and two F7MUX feed a F8MUX as shown in Figure C.4.

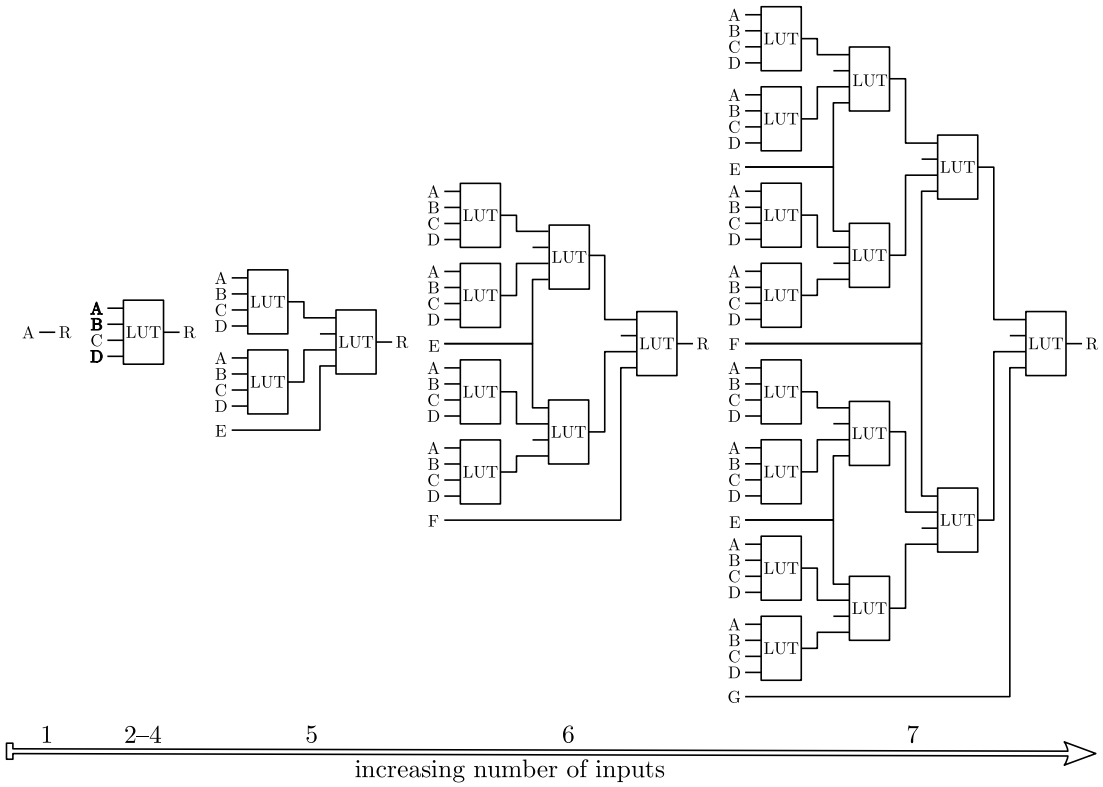


Figure C.2: Diagram showing how the cone of LUTs/logic grows exponentially with the number of inputs when using the multiplexer method.

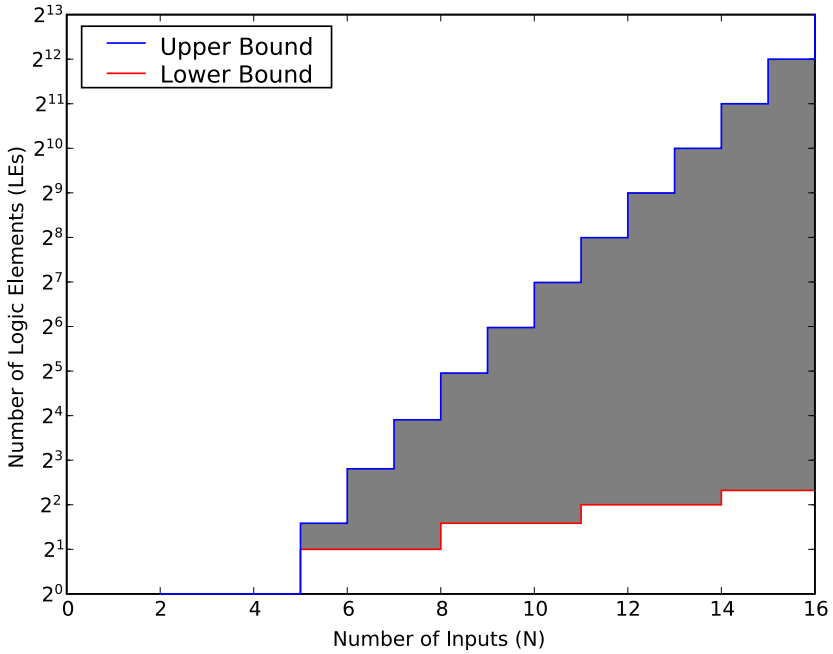


Figure C.3: A semilog plot of (C.1) and (C.2). The shaded area shows the range of LUTs required to implement an I input logic function.

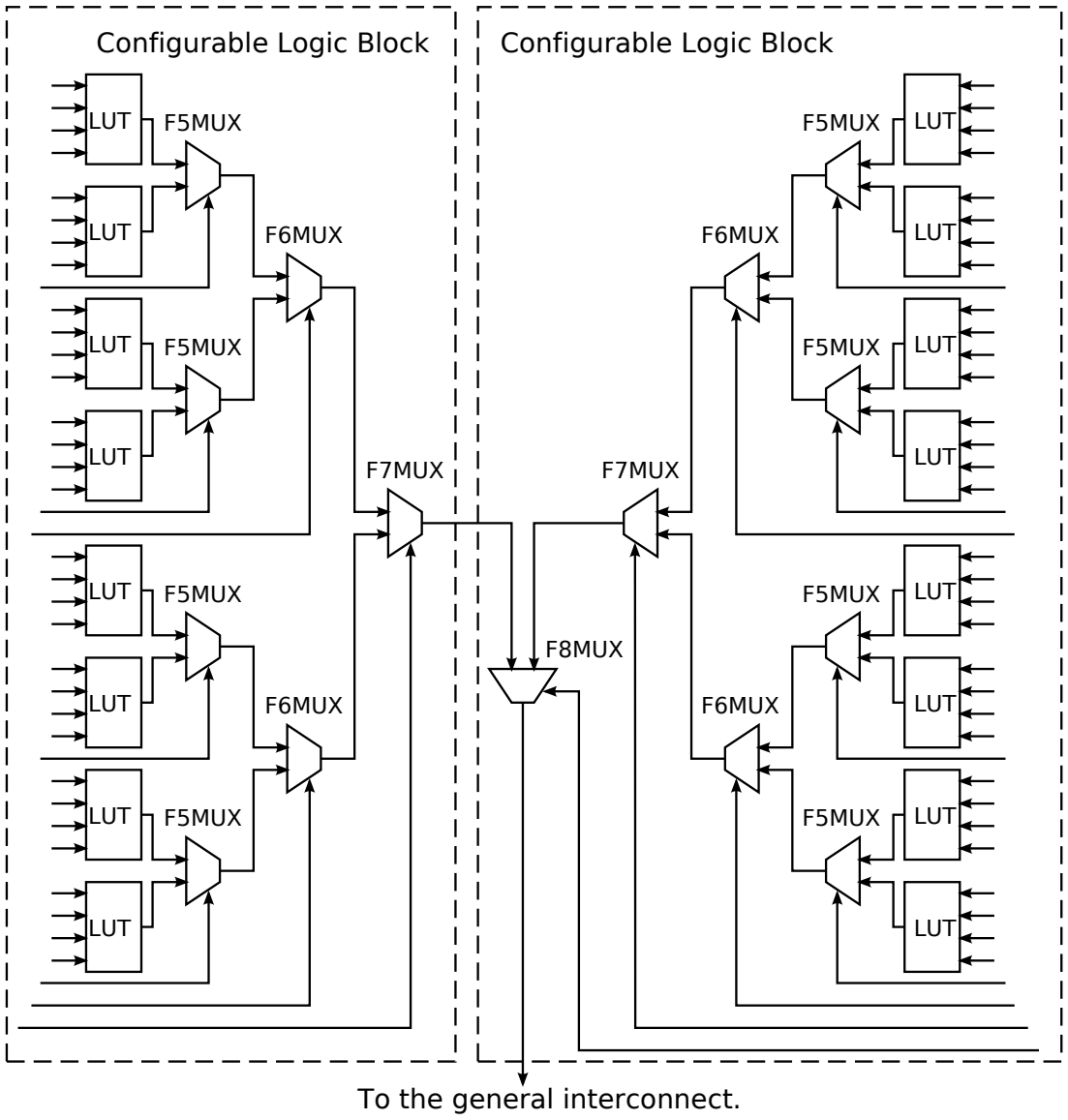


Figure C.4: A logic diagram showing the connections of the F*i*MUX within two Spartan 3 complex logic blocks. There is a total of 79 inputs; 64 inputs to the 16 LUTs and 15 inputs to the multiplexers.

As explained in section C.1 there are a couple of alternatives for reducing the logic. Using the F*i*MUX replaces the second and third level of LUTs in Figure C.1 and Figure C.2. This means any logic function of 8 inputs, a 32 to 1 multiplexer, or limited functions of up to 79 inputs (like bit by bit comparison) can be constructed using only 16 LUTs.

REFERENCES

A. Weinberger and J. L. Smith. A logic for high-speed addition. *National Bureau of Standards*, pages 3–12, 1958.

Altera Corporation. http://altera.com/buy/, March 2007. URL `http://altera.com/buy/`. eStore for Altera Products.

Altera Corporation. 2008 Annual Report, 2008.

Altera Cyclone III. *Cyclone III Device Handbook*. Altera Corporation, 2.1 edition, November 2008.

Altera Nios II. *Nios II Processor reference handbook*. Altera Corporation, 9.0 edition, March 2009.

Altera Primitives User Guide. *Designing with low-level primitives user guide*. Altera Corporation, 3.0 edition, April 2007.

Altera Stratix III. *Stratix III Device Handbook*. Altera Corporation, 1.7 edition, February 2009.

Ashok Ambardar. *Analogue and digital signal processing*. Brooks/Cole Publishing Company, 2 edition, 1999.

Amit Verma. How to design an FPGA architecture tailored for efficiency and performance. Technical report, Altera, 12 February 2007. URL `http://www.pldesignline.com/howto/architectureimplementation/197005332`.

S. Arno and F.S. Wheeler. Signed digit representation of minimum Hamming weight. *IEEE Transactions on Computers*, 42(8):1007–1010, August 1993.

A. Avizienis. Signed-digit number representation for fast parallel arithmetic. *IRE Trans. Elect. Comput.*, EC-10:389–400, September 1961.

A. A. Balandin. Chill Out. *IEEE Spectrum*, pages 35–39, October 2009.

Paul Beckett. Towards a balanced ternary FPGA. In *Proceedings of the 2009 International Conference on Field Programmable Technology*, pages 46–53, 9 December 2009.

G. Bergman. A number system with an irrational base. *Mathematics Magazine*, 31(2): 98–110, 1957.

G. W. Bewick. *Fast multiplication: algorithms and implementation.* PhD thesis, Stanford University, 1994.

U. Narayan Bhat. *An introduction to queueing theory : modeling and analysis in applications.* Birkhauser, 2008.

E. Blossom. GNU radio: tools for exploring the radio frequency spectrum. *Linux Journal*, 2004(122):4, 2004.

Andrew D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, pages 236–240, June 1951.

N. Boullis and A. Tisserand. Some optimizations of hardware multiplication by constant matricies. *IEEE Transactions on Computers*, 54(10):1271–1282, 10 October 2005.

R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31:260–264, March 1982.

S. Brown and Z. G. Vranesic. *Fundamentals of digital logic with verilog design.* McGraw-Hill, 1 edition, 2003.

A. P. Chandrakasan and R. W. Brodersen. Minimizing power consumption in digital CMOS circuits. *Proceedings of the IEEE*, 83(4):498–523, April 1995.

A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

Y.N. Chang and K.K. Parhi. High-performance digit-serial complex-number multiplier-accumulator. In *International conference on computer design: VLSI in computers and processors, 1998. ICCD'98. proceedings*, pages 211–213, 1998.

P. P. Chu. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability.* Wiley-IEEE Press, 2006.

Richard G. Cliff and L. Todd Cope. Look up table implementation of fast carry arithmetic and exclusive-OR operations, 13 December 1993. Reissued 2 Jan 1996.

Jeffrey O. Coleman and Arda Yurdakul. Fractions in canonical-signed-digit number system. In *2001 Conference on Information Sciences and Systems*, 21 March 2001.

W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatorial optimization.* John Wiley & Sons, Inc. New York, NY, USA, 1998.

L. Dadda. Some schemes for parallel multipliers. *Digital Signal Computers & Processors*, page 208, 1977.

L. D'Addario, A. Gunst, J. Bunton, and A. J. Boonstra. SKA signal processing. *SKA Memo 91*, page 98, August 2007.

I. Das and J. E. Dennis. Normal-boundary intersection: a new method for generating Pareto optimal points in multicriteria optimization problems. *SIAM Journal on Optimization*, 8(3):631–657, 1998.

A. G. Dempster and M. D. Macleod. Use of minimum-adder multiplier blocks in FIR digital filters. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, 42(9):569–577, 1995.

A. G. Dempster and M. D. Macleod. Generation of signed-digit representations for integer multiplication. *Signal Processing Letters, IEEE*, 11(8):663–665, 2004.

R. W. Doran. Variants of an improved carry look-ahead adder. *IEEE Transactions on Computers*, 37(9):1110–1113, 9 September 1988.

O. Egecioglu and C.K. Koc. Exponentiation using canonical recoding. *Science*, 129: 407–417, 1994.

M.D. Ercegovac and T. Lang. Fast multiplication without carry-propagate addition. *Computers, IEEE Transactions on*, 39(11):1385 –1390, November 1990.

M. T. Frederick and A. K. Somani. Multi-bit carry chains for high-performance reconfigurable fabrics. In *Int. Conf. Field Prog. Logic and Applications (FPL'06)(Madrid, Spain*, 2006.

A. F. González and P. Mazumder. Redundant arithmetic, algorithms and implementations. *Integration, The VLSI Journal*, 30(1):13–53, 2000.

D.M. Gordon. A survey of fast exponentiation methods. *Journal of algorithms*, 27(1): 129–146, 1998.

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete mathematics: a foundation for computer science*. Addison-Wesley Reading, MA, 1994.

H. Gundersen, R. Jensen, and Y. Berg. A novel ternary switching element using CMOS recharge semi floating-gate devices. In *Proc. 35th Int. Symp. Multiple-Valued Logic*, pages 54–58, 2005.

R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, and N. Takagi. A high-speed multiplier using a redundant binary adder tree. *IEEE Journal of Solid-State Circuits*, 22(1):28–34, 2 August 1987.

Richard I. Hartley. Subexpression sharing in filters using canonic signed digit multipliers. *IEEE Transactions on Circuits and Systems—Part II: Analog and Digital Signal Processing*, 43(10):677–688, 10 October 1996.

S. Hauck, M. M. Hosler, and T. W. Fry. High-performance carry chains for FPGA's. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2):138–147, 2000.

B. Hayes. Third base. *American Scientist*, 89(6):490–494, June 2001.

Simon Haykin. *Communication systems.* Wiley, 4th edition, May 2000.

O. Herrmann, L. R. Rabiner, and D. S. K. Chan. Practical design rules for optimum finite impulse response low-pass digital filters. *Bell Syst. Tech. J*, 52(6):769–799, 1973.

M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, pages 33–38, July 2008.

W.N. Holmes. Representation for complex numbers. *IBM Journal of Research and Development*, 22(4):429–430, 1978.

John Hopcroft. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Boston, 2001.

Huey Ling. High speed binary adder. *IBM Journal of Resarch Development*, 25(3): 156–166, May 1981.

K. Hwang. *Computer Arithmetic.* John Wiley & Sons, New York, 1979.

Emmanuel C. Ifeachor and Barrie W. Jervis. *Digital Signal Processing : A Practical Approach.* Prentice Education Limited, 2 edition, 2002.

J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, pages 226–231, June 1960.

G. Jaberipur and B. Parhami. Constant-time addition with hybrid-redundant numbers: Theory and implementations. *Integration, the VLSI Journal*, 41(1):49–64, 2008.

G. Jaberipur, B. Parhami, and M. Ghodsi. A class of stored-transfer representations for redundant number systems. In *Signals, Systems and Computers, 2001. Conference Record of the Thirty-Fifth Asilomar Conference on*, volume 2, pages 1304 –1308, 2001.

T. Jamil. The complex binary number system. *IEEE potentials*, 20(5):39–41, 2001.

J. Jedwab and CJ Mitchell. Minimum weight modified signed-digit representations and fast exponentiation. *Electronics Letters*, 25(17):1171–1172, 1989.

John F. Wakerly. *Digital Design: Principles and Practices.* Prentice-Hall, 2000.

D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python. *URL: http://www. scipy. org*, 2001.

William Kamp and Andrew Bainbridge-Smith. Multiply accumulate unit optimised for fast dot-product evaluation. *Field-Programmable Technology, 2007. International Conference on*, pages 349–352, 12 December 2007.

William Kamp, Andrew Bainbridge-Smith, and Michael Hayes. Efficient implementation of fast redundant number adders for long word-lengths in FPGA. *Field Programmable Technology, 2009, International Conference on*, 9 December 2009.

S. M. Kang and Y. Leblebici. *CMOS digital integrated circuits: analysis and design*. McGraw-Hill Science/Engineering/Math, 2002.

Shoji Kawahito, Michitaka Kameyama, and Tatsuo Higuchi. Multiple-valued radix-2 signed-digit arithmetic circuits for high-performance VLSI systems. 25(1), February 1990.

C. Kaya Koc, T. Acar, and BS Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE micro*, 16(3):26–33, 1996.

Walt Kester. *Mixed-signal and DSP design techniques*. Newnes, Analogue Devices Inc., 2003.

M. Khabbazian, T.A. Gulliver, and V.K. Bhargava. A new minimal average weight representation for left-to-right point multiplication methods. *IEEE transactions on computers*, pages 1454–1459, 2005.

MW Kharrat, B. Ayed, M. Loulou, N. Masmoudi, and L. Kamoun. A new method to implement a constant operand multiplier. In *Microelectronics, The 14th International Conference on 2002-ICM*, pages 62–65, 2002.

T. Kilburn, D. B. G. Edwards, and D. Aspinall. Parallel addition in digital computers: A new fast carry circuit. *IEE*, 106:464–466, October 1959.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

D.E. Knuth. A imaginary number system. *Communications of the ACM*, 3(4):247, 1960.

Donald Knuth. *Positional number systems*, volume 2 of *The art of computer programming*. Addison-Wesley, Boston, 3 edition, 1998.

C. K. Koc. Parallel canonical recoding. *Electronic Letters*, 32(22):2063–2065, 24 October 1996.

C. K. Koc and C. Hung. Adaptive m-ary segmentation and canonical recoding algorithms for multiplication of large binary integers. *Computers and Math. with Applications*, 24(3):3–12, March 1992.

Cetin Kaya Koc and Scott Johnson. Multiplication of signed-digit numbers. *Electronics Letters*, 30(11):840–841, 26 May 1994.

P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

I. Koren. *Computer arithmetic algorithms*. AK Peters, Ltd., 2 edition, 2002.

S. Kuninobu, T. Nishiyama, H. Edamatsu, and T. Taniguchi. Design of high speed MOS multiplier and divider using redundant binary representation. In *8th IEEE Symposium on Computer Arithmetic*, 19 May 1987a.

S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi. Design of high speed MOS multiplier and divider using redundant binary representation. In *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, pages 80–86, 1987b.

Ian Kuon and Jonathan Rose. Area and delay trade-offs in the circuit and architecture design of FPGAs. In *FPGA'08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 149–158, New York, NY, USA, 2008. ACM.

E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966. ISSN 0030364X.

D.C. Lou and C.L. Wu. Parallel modular exponentiation using signed-digit-folding technique. *INFORMATICA-LJUBLJANA-*, 28(2):197–206, 2004.

G.K. Ma and F.J. Taylor. Multiplier policies for digital signal processing. *IEEE Assp Magazine*, 7(1):6–20, 1990.

T. E. Mangir. Sources of failures and yield improvement for vlsi and restructurable interconnects for rvlsi and wsi: Part i—sources of failures and yield improvement for vlsi. *Proceedings of the IEEE*, 72(6):690–708, June 1984.

M. Morris Mano and Charles R. Kime. *Logic and Computer Design Fundamentals*. Prentice Hall, 3 edition, 2004.

J. McClellan, T. Parks, and L. Rabiner. A computer program for designing optimum fir linear phase digital filters. *Audio and Electroacoustics, IEEE Transactions on*, 21 (6):506–526, Dec 1973. ISSN 0018-9278.

W. Mikhael and F. Wu. Fast algorithms for block fir adaptive digital filtering. *Circuits and Systems, IEEE Transactions on*, 34(10):1152–1160, October 1987.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, 19 April 1965.

F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Theoretical Informatics and Applications*, 24:531–544, 1990.

J. A. Muir and D. R. Stinson. Minimality and other proprieties of the widthw nonadjacent form. *University of Waterloo, Canada*, 2004.

J. A. Muir and D. R. Stinson. New minimal weight representations for left-to-right window methods. *Topics in Cryptology – CT-RSA 2005*, 3376:336 – 383, 2005.

C. Nagendra, M. J. Irwin, R. M. Owens, C. M. Div, A. M. D. Inc, and C. A. Sunnyvale. Area-time-power tradeoffs in parallel adders. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on]*, 43(10):689–702, 1996.

S. H. A. Niaki, A. Cevrero, P. Brisk, C. Nicopoulos, F. K. Gurkaynak, Y. Leblebici, and P. Ienne. Design space exploration for field programmable compressor trees. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 207–216. ACM New York, NY, USA, 2008.

O. J. Bedrij. Carry-select adder. *IRE Transactions on Electronic Computers*, pages 340–346, June 1962.

AM Odlyzko. Non-negative digit sets in positional number systems. *Proceedings of the London Mathematical Society*, 3(2):213, 1978.

A.O. Ogunfunmi and A.M. Peterson. Fast direct implementation of block adaptive fir filtering. In *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*, pages 920–923 vol.2, May 1989.

V.G. Oklobdzija and M.D. Ercegovac. An on-line square root algorithm. *Computers, IEEE Transactions on*, C-31(1):70 –75, January 1982.

H. Parandeh-Afshar, P. Brisk, and P. Ienne. A novel FPGA logic block for improved arithmetic performance. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 171–180. ACM New York, NY, USA, 2008.

B. Parhami. Generalized signed-digit number systems: a unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39(1):89–98, 1990.

B. Parhami. *Computer arithmetic*. Oxford University Press, 2000.

Behrooz Parhami. Carry-free addition of recoded binary signed-digit numbers. *IEEE Transactions on Computers*, 37(11):1470–1476, 11 November 1988.

Dhanajay S. Phatak and I. Koren. Hybrid signed-digit number systems: A unified framework for redundant number representations with bounded carry propagation chains. *IEEE Transactions on Computers*, 43(8):880–891, 8 August 1994.

D.S. Phatak, T. Goff, and I. Koren. Constant-time addition and simultaneous format conversion based on redundant binary representations. *IEEE Transactions on Computers*, 50(11):1267–1278, November 2001.

B. Phillips and N. Burgess. Minimal weight digit set conversions. *IEEE Transactions on Computers*, 53(6):666–677, June 2004.

D. Poole. *Linear algebra: a modern introduction*. Thomson Brooks/Cole, 2 edition, 2006.

J. R. Powell. The quantum limit to Moore's law. *Proceedings of the IEEE*, 96(8): 1247–1248, August 2008.

Altera Quartus II Handbook. *Quartus II Development Software Handbook*. Altera Corporation, 9.0 edition, March 2009.

G.W. Reitwiesner. Binary arithmetic. *Advances in computers*, 1:231–308, 1960.

Homayoon Sam and Arupratan Gupta. A generalized multibit recoding of two's complement binary numbers and it's proof with application in multiplier implementations. *IEEE Transactions on Computers*, 39(8):1006–1015, 8 August 1990.

H. Samueli. An improved search algorithm for the design of multiplierless FIR filters with powers-of-two coefficients. *Circuits and Systems, IEEE Transactions on*, 36(7): 1044–1047, May 1989.

P. Sedcole and P. Y. K. Cheung. Parametric yield in FPGAs due to within-die delay variations: a quantitative analysis. In *FPGA 2007: Fifteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 178. Association for Computing Machinery, 18 February 2007.

K. G. Smitha, H. A. H. Fahmy, and A. P. Vinod. Redundant adders consume less energy. *Circuits and Systems, IEEE Asia Pacific Conference on*, pages 422–425, 2006.

M. A. Soderstrand, L. G. Johnson, H. Arichanthiran, M. D. Hoque, and R. Elangovan. Reducing hardware requirement in FIR filter design. *Acoustics, Speech, and Signal Processing, 2000. ICASSP'00. Proceedings. 2000 IEEE International Conference on*, 6, 2000.

F. X. Standaert. Introduction to side-channel attacks. Technical report, UCL Crypto Group, Belgium, 6 February 2009. URL `http://www.dice.ucl.ac.be/~fstandae/PUBLIS/42.pdf`.

P. Suppes. *Introduction to logic*. Dover Publications, 1999.

N. Takagi, H. Yasuura, and S. Yajima. High-speed VLSI multiplication algorithm with a redundant binary addition tree. *IEEE Transactions on Computers*, 100(34):789–796, 1985.

R. Tarjan. Depth-first search and linear graph algorithms. In *Switching and automata theory, 12th annual symposium on*, pages 114–121, 1971.

K. Tatas, G. Koutroumpezis, D. Soudris, and A. Thanailakis. Architecture design of a coarse-grain reconfigurable multiply-accumulate unit for data-intensive applications. *Integration, the VLSI Journal*, 40(2):74–93, February 2007.

H. J. M. Veendrick. Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits. *Solid-State Circuits, IEEE Journal of*, 19(4):468–473, August 1984.

K. Vitoroulis and A.J. Al-Khalili. Performance of parallel prefix adders implemented with FPGA technology. In *Circuits and Systems, 2007. NEWCAS 2007. IEEE Northeast Workshop on*, pages 498–501, August 2007.

Yevgen Voronenko and Markus Püschel. Multiplierless multiple constant multiplication. *ACM Transactions on Algorithms*, 3(2):11, May 2007.

C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, pages 14–17, 1964.

Y. Wang and K. Roy. CSDC: A new complexity reduction technique for multiplierless implementation of digital FIR filters. *Circuits and Systems I: Regular Papers, IEEE Transactions on [see also Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on]*, 52(9):1845–1853, 2005.

Huapeng Wu and M. Anwar Hasan. Closed-form expression for the average weight of signed-digit representations. *IEEE Transactions on Computers*, 48(8):848–851, 8 August 1999.

Xilinx Microblaze. *Microblaze processor reference guide.* Xilinx, Inc., 9.0 edition, 17 January 2008.

Xilinx Spartan-3. *Spartan-3 Generation FPGA User Guide.* Xilinx, Inc., ug331 v1.3 edition, 14 February 2008.

Xilinx Virtex-5. *Virtex-5 FPGA User Guide.* Xilinx, Inc., 4.5 edition, January 2009.

F. Xu, C. H. Chang, and C. C. Jong. Hamming weight pyramid – A new insight into canonical signed digit representation and its applications. *Computers and Electrical Engineering*, 33(3):195–207, 2007.

S. M. Yen, C. S. Laih, C. H. Chen, and J. Y. Lee. An efficient redundant-binary number to binary number converter. *Solid-State Circuits, IEEE Journal of*, 27(1):109–112, 1992.